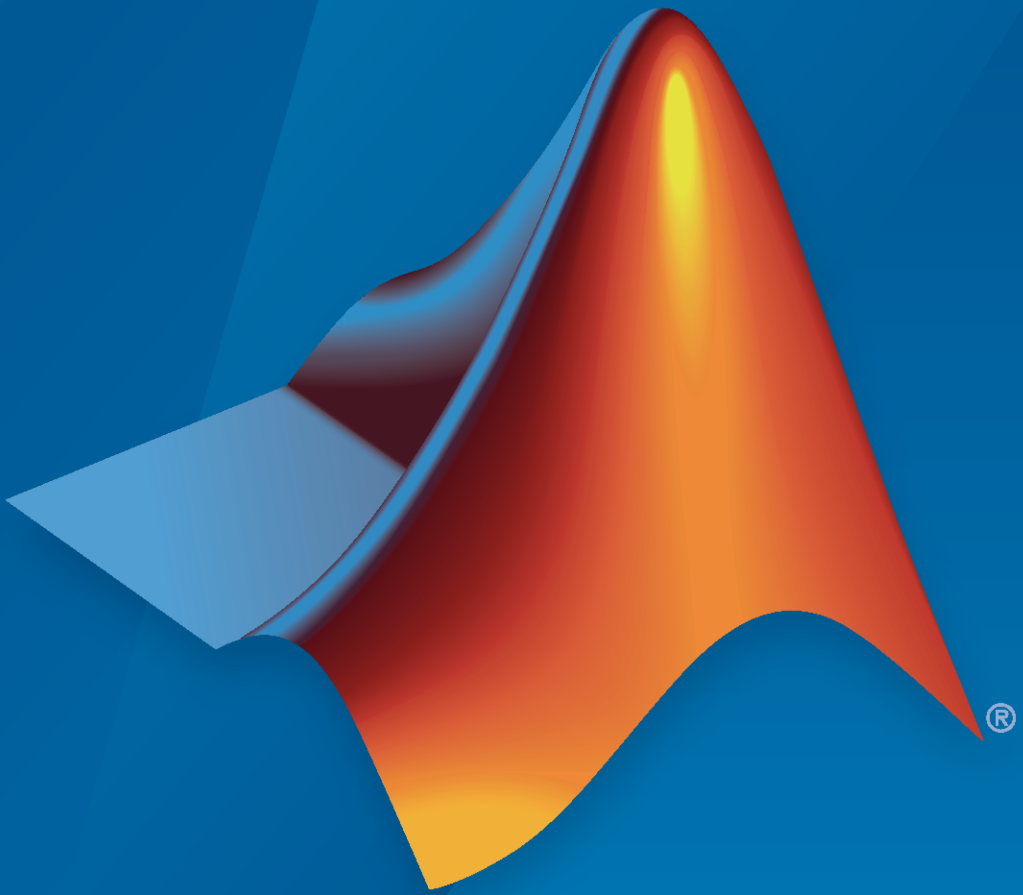
Financial Toolbox™

User's Guide



MATLAB®

R2023a

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.
1 Apple Hill Drive
Natick, MA 01760-2098

Financial Toolbox™ User's Guide

Trademarks

Patents

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)
March 2018	Online only	Revised for Version 5.11 (Release 2018a)
September 2018	Online only	Revised for Version 5.12 (Release 2018b)
March 2019	Online only	Revised for Version 5.13 (Release 2019a)
September 2019	Online only	Revised for Version 5.14 (Release 2019b)
March 2020	Online only	Revised for Version 5.15 (Release 2020a)
September 2020	Online only	Revised for Version 6.0 (Release 2020b)
March 2021	Online only	Revised for Version 6.1 (Release 2021a)
September 2021	Online only	Revised for Version 6.2 (Release 2021b)
March 2022	Online only	Revised for Version 6.3 (Release 2022a)
September 2022	Online only	Revised for Version 6.4 (Release 2022b)
March 2023	Online only	Revised for Version 6.5 (Release 2023a)

Contents

Getting Started

1

Performing Common Financial Tasks

2

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 Using Financial Timetables

12 Trading Date Utilities

13 Technical Analysis

14 Stochastic Differential Equations

15 Functions

A Bibliography

1

Getting Started

Financial Toolbox Product Description

Analyze financial data and develop financial models

Financial Toolbox provides functions for the mathematical modeling and statistical analysis of financial data. You can analyze, backtest, and optimize investment portfolios taking into account turnover, transaction costs, semi-continuous constraints, and minimum or maximum number of assets. The toolbox enables you to estimate risk, model credit scorecards, analyze yield curves, price fixed-income instruments and European options, and measure investment performance.

Stochastic differential equation (SDE) tools let you model and simulate a variety of stochastic processes. Time series analysis functions let you perform transformations or regressions with missing data and convert between different trading calendars and day-count conventions.

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

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 1-15

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, *Macro-Investment Analysis* by William Sharpe also provides an excellent explanation of matrix algebra operations using MATLAB. It is available on the web at:

```
https://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 this is meaningless for this example.

Computing Dot Products of Vectors

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

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

then the dot product

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

is the scalar product of the two vectors. It is an exception to the commutative rule. To compute the dot product in MATLAB, use `sum(X .* Y)` or `sum(Y .* X)`. 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 *i*th row and *j*th 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 previously mentioned 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 1-11). 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 ≠ 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

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

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

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

where *A**X* = *B*.

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

```
A = [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 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 1-15

Using Input and Output Arguments with Functions

In this section...

Input Arguments

Vector and Matrix Input

By design, MATLABsoftware can efficiently perform repeated operations on collections of data stored in vectors and matrices. MATLAB code that is written to operate simultaneously on different arrays is said to be vectorized. Vectorized code is not only clean and concise, but is also efficiently processed by MATLAB.

Because MATLAB is optimized for processing vectorized code, many Financial Toolbox functions accept either vector or matrix input arguments, rather than single (scalar) values.

One example of such a function is the `irr` function, which computes the internal rate of return of a cash flow stream. If you input a vector of cash flows from a single cash flow stream, then `irr` returns a scalar rate of return. If you input a matrix of cash flows from multiple cash flow streams, where each matrix column represents a different stream, then `irr` returns a vector of internal rates of return, where the columns correspond to the columns of the input matrix. Many other Financial 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 is stored in a vector

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

```
CashFlows =
  -100
    10
    20
    30
    40
    50
```

Use the `irr` function to compute the internal rate of return of the cash flow stream.

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

```
Rate =

    0.1201
```

For the single cash flow stream `CashFlows`, the function returns a scalar rate of return of `0.1201`, or 12.01%.

Now, use the `irr` function to compute internal rates of return for multiple cash flow streams.

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

    0.1201    0.1201    0.1201
```

MATLAB performs the same computation on all the assets at once. For the three cash flow streams, the `irr` function returns a vector of three internal rates of return.

In the Financial Toolbox context, vectorized programming is useful in portfolio management. You can organize multiple assets into a single collection by placing data for each asset in a different matrix column or row, then pass the matrix to a Financial Toolbox function.

Character Vector Input

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

A character vector is stored as a character array, one ASCII character per element. Thus, the date character vector is

```
DateCharacterVector = '9/16/2017'
```

This date character vector is actually a `1`-by-`9` vector. If you create a vector or matrix of character vectors, each character vector must have the same length. Using a column vector to create a vector of character vectors can allow you to visually check that all character vectors are the same length. If your character vectors are not the same length, use spaces or zeros to make them the same length, as in the following code.

```
DateFields = ['01/12/2017'
              '02/14/2017'
              '03/03/2017'
              '06/14/2017'
              '12/01/2017'];
```

`DateFields` is a `5`-by-`10` array of character vectors.

You cannot mix numbers and character vectors in a vector or matrix. If you input a vector or matrix that contains a mix of numbers and character vectors, MATLAB treats every entry as a character. As an example, input the following code

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

```
Item =

SZc14-Sep-1999
```

The software understands the input not as a `1`-by-`4` vector, but as a `1`-by-`14` character array with the value `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(input_arguments...)
```

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(input_arguments...)
```

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

Some functions that return vectors accept only scalars as arguments. Such functions cannot accept vectors as arguments and return matrices, where each column in the output matrix corresponds to an entry in the input. Output vectors can be variable length.

For example, most functions that require asset life as an input, and return values corresponding to different periods over the asset life, cannot handle vectors or matrices as input arguments. These functions include `amortize`, `depfixdb`, `depgendb`, and `depsoyd`. For example, consider a car for which you want to compute the depreciation schedule. Use the `depfixdb` function to compute a stream of declining-balance depreciation values for the asset. Set the initial value of the asset and the lifetime of the asset. Note that in the returned vector, the asset lifetime determines the number of rows. Now consider a collection of cars with different lifetimes. Because `depfixdb` cannot output a matrix with an unequal number of rows in each column, `depfixdb` cannot accept a single input vector with values for each asset in the collection.

See Also

Related Examples

- “Matrices and Arrays”

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 preserved. For example:

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


b = 

  datetime

   2021-05-04 15:59:34
```

- 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”.

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 Text and datetime or duration Values”.

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 “Character Vector Input” on page 1-16.

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

`datedisp` | `datenum` | `datestr` | `datetime` | `datevec` | `format` | `date` | `holidays` | `nyseclosures` | `busdate` | `cfdates` | `addBusinessCalendar`

Related Examples

- “Convert Between Text and datetime or duration Values”
- “Read Collection or Sequence of Spreadsheet Files”
- “Trading Calendars User Interface” on page 12-2
- “UICalendar User Interface” on page 12-4

More About

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

External Websites

- Automated Data Cleaning and Preparation in MATLAB (43 min)

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)
```

```
ROR =

    0.1172
```

The rate of return is 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)
```

```
Rate =

    0.0938
```

The Rate 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)
```

```
NPV =

   5.9117e-12
```

The NPV is 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)
```

```
Rate =

    0.0094
```

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)
```

```
Principal =

   1.1607e+04
```

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

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

Related Examples

- “Handle and Convert Dates” on page 2-2
- “Pricing and Computing Yields for Fixed-Income Securities” on page 2-15

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). Financial Instruments Toolbox™ supports additional functionality for pricing fixed-income securities. For more information, see “Price Interest-Rate Instruments” (Financial Instruments Toolbox).

Fixed-Income Terminology

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

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.

Period

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

Basis

The basis of a bond refers to the basis or day-count convention for a bond. Day count basis determines how interest accrues over time for various instruments and the amount transferred on interest payment dates. 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 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. Also, another common actual/actual basis is basis `12`.

Basis Value	Day Count Convention
1	30/360 SIA — Year fraction is calculated based on a 360 day year with 30-day months, after applying the following rules: If the first date and the second date are the last day of February, the second date is changed to the 30th. If the first date falls on the 31st or is the last day of February, it is changed to the 30th. If after the preceding test, the first day is the 30th and the second day is the 31st, then the second day is changed to the 30th.
2	actual/360 — Number of days in a period is equal to the actual number of days, however the number of days in a year is 360.
3	actual/365 — Number of days in a period is equal to the actual number of days, however the number of days in a year is 365 (even in a leap year).
4	30/360 PSA — Number of days in every month is set to 30 (including February). If the start date of the period is either the 31st of a month or the last day of February, the start date is set to the 30th, while if the start date is the 30th of a month and the end date is the 31st, the end date is set to the 30th. The number of days in a year is 360.
5	30/360 ISDA — Number of days in every month is set to 30, except for February where it is the actual number of days. If the start date of the period is the 31st of a month, the start date is set to the 30th while if the start date is the 30th of a month and the end date is the 31st, the end date is set to the 30th. The number of days in a year is 360.
6	30E /360 — Number of days in every month is set to 30 except for February where it is equal to the actual number of days. If the start date or the end date of the period is the 31st of a month, that date is set to the 30th. The number of days in a year is 360.
7	actual/365 Japanese — Number of days in a period is equal to the actual number of days, except for leap days (29th February) which are ignored. The number of days in a year is 365 (even in a leap year).
8	actual/actual ICMA — Number of days in both a period and a year is the actual number of days and the compounding frequency is annual.
9	actual/360 ICMA — Number of days in a period is equal to the actual number of days, however the number of days in a year is 360 and the compounding frequency is annual.
10	actual/365 ICMA — Number of days in a period is equal to the actual number of days, however the number of days in a year is 365 (even in a leap year) and the compounding frequency is annual.
11	30/360 ICMA — Number of days in every month is set to 30, except for February where it is equal to the actual number of days. If the start date or the end date of the period is the 31st of a month, that date is set to the 30th. The number of days in a year is 360 and the compounding frequency is annual.
12	actual/365 ISDA — The day count fraction is calculated using the following formula: `(Actual number of days in period that fall in a leap year / 366)` + `(Actual number of days in period that fall in a normal year / 365)`. Basis 12 is also referred to as actual/actual ISDA.
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.

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.

End-of-Month Rule

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)
```

```
Price =

  113.5977


AccruedInt =

    0.5855
```

bndprice 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 `bndyield` function

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

```
Yield =

    0.0659
```

The function returns a `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). The Macaulay duration is a measure of price sensitivity to yield changes. This duration is measured in years and is a weighted average-time-to-maturity of an instrument.

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

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

```
ModDuration =

    1.6107


YearDuration =

    1.6470


PerDuration =

    3.2940
```

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 https://www.treasury.gov/resource-center/data-chart-center/interest-rates/Pages/TextView.aspx?data=yield:

```
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
- “Term Structure of Interest Rates” on page 2-29
- “Computing Treasury Bill Price and Yield” on page 2-26

More About

- “Treasury Bills Defined” on page 2-25

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` | `tbillyield` | `tbillyield2disc` | `tbillval01` | `tbl2bond` | `tr2bonds` | `zbtprice` | `zbtyield`

Related Examples

- "Handle and Convert Dates" on page 2-2
- "Term Structure of Interest Rates" on page 2-29
- "Computing Treasury Bill Price and Yield" on page 2-26

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 (one hundredth of one percentage point, or 0.0001) 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` | `tbillyield` | `tbillyield2disc` | `tbillval01` | `tbl2bond` | `tr2bonds` | `zbtprice` | `zbtyield`

Related Examples

- “Handle and Convert Dates” on page 2-2
- “Term Structure of Interest Rates” on page 2-29

More About

- “Treasury Bills Defined” on page 2-25

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 =

   1.0e+05 *

         0    7.3085    0.0010    0.0010    0.0000
```

(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 =

   1.0e+05 *

         0    7.3085    0.0010    0.0010    0.0000
    0.0000    7.3116    0.0010    0.0010    0.0000
```

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)
```

```
Bonds =

   1.0e+05 *

    7.3085         0    0.0010         0         0    0.0000
    7.3116    0.0000    0.0010    0.0000         0    0.0000


Prices =

   99.2654
  103.1875


Yields =

    0.0510
    0.0564
```

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
```

Use additional Financial Toolbox functions `zero2disc`, `zero2fwd`, and `zero2pyld` to 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` | `tbillyield` | `tbillyield2disc` | `tbillval01` | `tbl2bond` | `tr2bonds` | `zbtprice` | `zbtyield`

Related Examples

- “Handle and Convert Dates” on page 2-2
- “Computing Treasury Bill Price and Yield” on page 2-26

More About

- “Treasury Bills Defined” on page 2-25

Returns with Negative Prices

Once considered a mathematical impossibility, negative prices have become an established aspect of many financial markets. Negative prices arise in situations where investors determine that holding an asset entails more risk than the current value of the asset. For example, energy futures see negative prices because of costs associated with overproduction and limited storage capacity. In a different setting, central banks impose negative interest rates when national economies become deflationary, and the pricing of interest rate derivatives, traditionally based on positivity, have to be rethought (see “Work with Negative Interest Rates Using Functions” (Financial Instruments Toolbox)). A negative price encourages the buyer to take something from the seller, and the seller pays a fee for the service of divesting.

MathWorks® Computational Finance products support several functions for converting between price series *p*(*t*) and return series *r*(*t*). Price positivity is not a requirement. The returns computed from input negative prices can be unexpected, but they have mathematical meaning that can help you to understand price movements.

Negative Price Conversion

Financial Toolbox functions `ret2tick` and `tick2ret` support converting between price series *p*(*t*) and return series *r*(*t*).

For simple returns (default), the functions implement the formulas

$$r_s(t) = \frac{p_s(t)}{p_s(t-1)} - 1$$

$$p_s(t) = p_s(t-1)(r_s(t) + 1)\,.$$

For continuous returns, the functions implement the formulas

$$r_c(t) = \log\left(\frac{p_c(t)}{p_c(t-1)}\right)$$

$$p_c(t) = p_c(t-1)e^{r_c(t)}\,.$$

The functions `price2ret` and `ret2price` implement the same formulas, but they divide by Δ*t* in the return formulas and they multiply by Δ*t* in the price formulas. A positive factor of Δ*t* (enforced by required monotonic observation times) does not affect the behavior of the functions. Econometrics Toolbox™ calls simple returns *periodic*, and continuous returns are the default. Otherwise, the functionality between the set of functions is identical. This example concentrates on the Financial Toolbox functions.

In the simple return formula, $r_s(t)$ is the percentage change (*PC*) in $p_s(t-1)$ over the interval $[t-1,t]$

$$PC = \frac{p_s(t)}{p_s(t-1)} - 1$$

$$p_s(t) = p_s(t-1) + PC \cdot p_s(t-1)\,.$$

For positive prices, the range of *PC* is $(-1,\infty)$, that is, anything from a 100% loss (p_s: $p_s(t-1) \rightarrow 0$) to unlimited gain. The recursion in the second equation gives the subsequent prices; $p_s(t)$ is computed from $p_s(t-1)$ by adding a percentage of $p_s(t-1)$.

Furthermore, you can aggregate simple returns through time using the formula

$$\frac{p_s(T)}{p_s(1)} - 1 = \prod_{t=2}^{T} (r_s(t) + 1) - 1,$$

where the left-hand side represents the simple return over the entire interval $[0,T]$.

Continuous returns add 1 to PC to move the range to $(0,\infty)$, the domain of the real logarithm. Continuous returns have the time aggregation property

$$\log\left(\frac{p_c(T)}{p_c(1)} - 1\right) = \log\left(\prod_{t=2}^{T}\left(\frac{p_c(t)}{p_c(t-1)}\right)\right) = \sum_{t=2}^{T}\log\left(\frac{p_c(t)}{p_c(t-1)}\right) = \sum_{t=2}^{T} r_c(t)\,.$$

This transformation ensures additivity of compound returns.

If negative prices are allowed, the range of simple returns PC expands to $(-\infty,\infty)$, that is, anything from unlimited loss to unlimited gain. In the formula for continuous returns, logarithms of negative numbers are unavoidable. The logarithm of a negative number is not a mathematical problem because the complex logarithm (the MATLAB default) interprets negative numbers as complex numbers with phase angle $\pm\pi$, so that, for example,

$$-2 = 2e^{i\pi}$$

$$\log(- 2) = 2 + i\pi$$

If $x < 0$, $\log(x) = \log(|x|) \pm i\pi$. The log of a negative number has an imaginary part of $\pm\pi$. The log of 0 is undefined because the range of the exponential $e^{i\theta}$ is positive. Therefore, zero prices (that is, free exchanges) are unsupported.

Analysis of Negative Price Returns

To illustrate negative price inputs, consider the following price series and its simple returns.

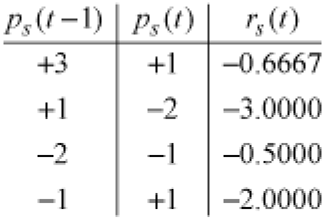
```
p = [3; 1; -2; -1; 1]
```

```
p =

     3
     1
    -2
    -1
     1
```

```
rs = tick2ret(p)
```

```
rs =

   -0.6667
   -3.0000
   -0.5000
   -2.0000
```

This table summarizes the recursions.



$p_s(t-1)$	$p_s(t)$	$r_s(t)$
+3	+1	−0.6667
+1	−2	−3.0000
−2	−1	−0.5000
−1	+1	−2.0000

The returns have the correct size (66%, 300%, 50%, 200%), but do they have the correct sign? If you interpret negative returns as losses, as with the positive price series, the signs seem wrong—the last two returns should be gains (that is, if you interpret less negative to be a gain). However, if you interpret the negative returns by the formula

$$p_S(t) = p_S(t - 1) + PC \cdot p_S(t - 1),$$

which requires the signs, the last two negative percentage changes multiply *negative* prices $p_s(t - 1)$, which produces *positive* additions to $p_s(t - 1)$. Briefly, a negative return on a negative price produces a positive price movement. The returns are correct.

The round trip produced by `ret2tick` returns the original price series.

```
ps = ret2tick(rs,StartPrice=3)
```

```
ps =

     3
     1
    -2
    -1
     1
```

Also, the following computations shows that time aggregation holds.

```
p(5)/p(1) - 1
```

```
ans =

   -0.6667
```

```
prod(rs + 1) - 1
```

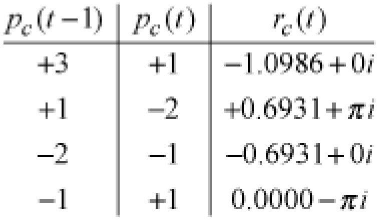
```
ans =

   -0.6667
```

For continuous returns, negative price ratios $p_c(t)/p_c(t - 1)$ are interpreted as complex numbers with phase angles ±*π*, and the complex logarithm is invoked.

```
rc = tick2ret(p,Method="continuous")
```

```
rc =

  -1.0986 + 0.0000i
   0.6931 + 3.1416i
  -0.6931 + 0.0000i
   0.0000 - 3.1416i
```

This table summarizes the recursions.



$p_c(t-1)$	$p_c(t)$	$r_c(t)$
+3	+1	$-1.0986+0i$
+1	−2	$+0.6931+\pi i$
−2	−1	$-0.6931+0i$
−1	+1	$0.0000-\pi i$

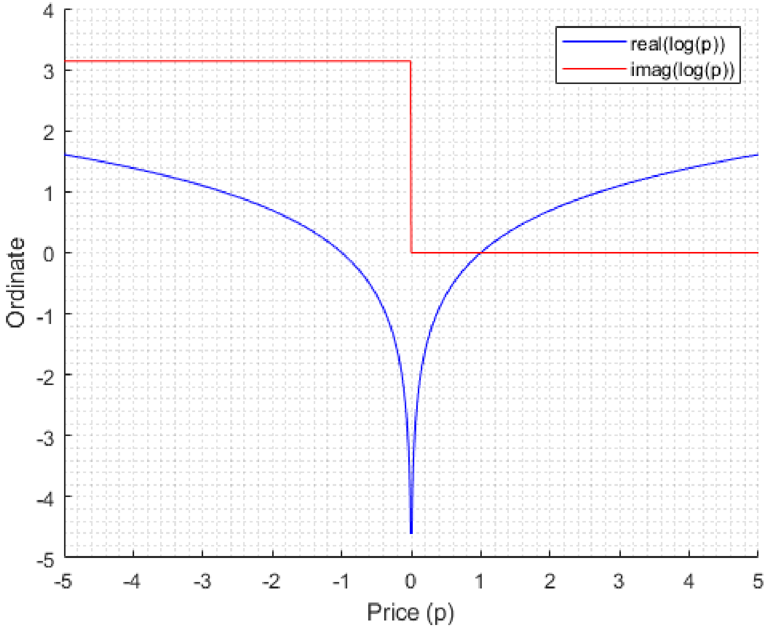
The real part shows the trend in the absolute price series. When $|p_c(t-1)| < |p_c(t)|$, that is, when prices move away from zero, $r_c(t)$ has a positive real part. When $|p_c(t-1)| > |p_c(t)|$, that is, when prices move toward zero, $r_c(t)$ has a negative real part. When $|p_c(t-1)| = |p_c(t)|$, that is, when the absolute size of the prices is unchanged, $r_c(t)$ has a zero real part. For positive price series, where the absolute series is the same as the series itself, the real part has its usual meaning.

The imaginary part shows changes of sign in the price series. When $p_c(t-1) > 0$ and $p_c(t) < 0$, that is, when prices move from investments to divestments, $r_c(t)$ has a positive imaginary part ($+\pi$). When $p_c(t-1) < 0$ and $p_c(t) > 0$, that is, when prices move from divestments to investments, $r_c(t)$ has a negative imaginary part ($-\pi$). When the sign of the prices is unchanged, $r_c(t)$ has a zero imaginary part. For positive price series, changes of sign are irrelevant, and the imaginary part conveys no information (0).

Visualization of Complex Returns

Complex continuous returns contain a lot of information. Visualizing the information can help you to interpret the complex returns. The following code plots the real and imaginary parts of the logarithm on either side of zero.

```
p = -5:0.01:5;
hold on
plot(p,real(log(p)),"b")
plot(p,imag(log(p)),"r")
xticks(-5:5)
xlabel("Price (p)")
ylabel("Ordinate")
legend(["real(log(p))" "imag(log(p))"],AutoUpdate=false)
grid minor
```



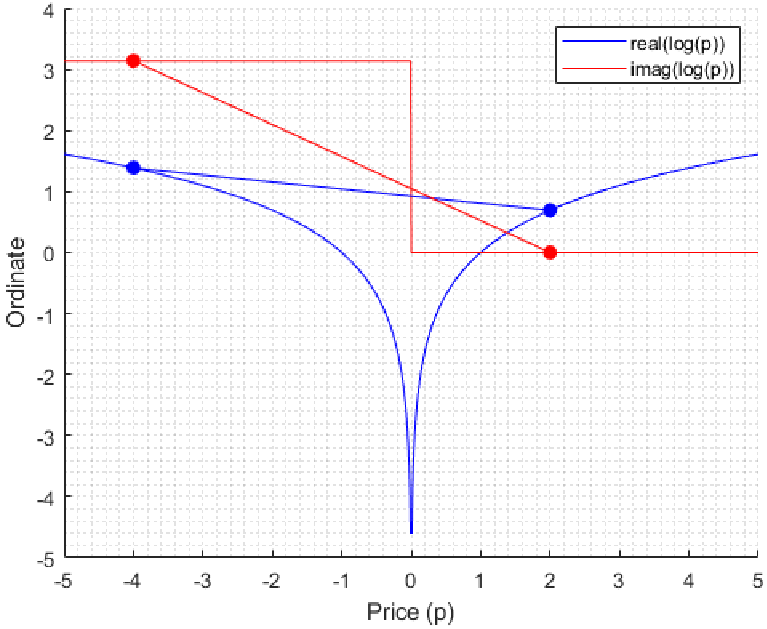
Due to the following identity

$$
\begin{aligned}
r_c(t) &= \log\left(\frac{p_c(t)}{p_c(t-1)}\right) \\
&= \log(p_c(t)) - \log(p_c(t-1)) \\
&= \underbrace{[\mathrm{real}(\log(p_c(t)) - \mathrm{real}(\log(p_c(t-1)))]}_{\text{blue curve}} + \underbrace{[\mathrm{imag}(\log(p_c(t))) - \mathrm{imag}(\log(p_c(t-1)))]}_{\text{red curve}} \cdot i,
\end{aligned}
$$

you can read the real part of a continuous return as a difference in ordinates on the blue graph, and you can read the imaginary part as a difference in ordinates on the red graph. Absolute price movements toward zero result in a negative real part and absolute price movements away from zero result in a positive real part. Likewise, changes of sign result in a jump of $\pm\pi$ in the imaginary part, with the sign change depending on the direction of the move.

For example, the plot below superimposes the real and imaginary parts of the logarithm at prices $p = -4$ and $p = 2$, with lines to help visualize their differences.

```
p = [-4; 2];
plot(p,real(log(p)),"bo-",MarkerFaceColor="b")
plot(p,imag(log(p)),"ro-",MarkerFaceColor="r")
hold off
```



If $p_c(t-1) = -4$ and $p_c(t) = 2$, the real part of $\log(p_c(t)) - \log(p_c(t-1))$ is negative (line slopes down), corresponding to a decrease in absolute price. The imaginary part is $0 - \pi = -\pi$, corresponding to a change of sign from negative to positive. If the direction of the price movement is reversed, so that $p_c(t-1) = 2$ and $p_c(t) = -4$, the positive difference in the real part corresponds to an increase in absolute price, and the positive difference in the imaginary part corresponds to a change of sign from positive to negative.

If you convert the continuous returns $r_c(t) = \log(p_c(t)/p_c(t-1))$ to simple returns $r_s = (p_s(t)/p_s(t-1) - 1)$ by the following computation, the result is the same simple returns series as before.

```
rs = exp(rc) - 1
```

```
rs =

  -0.6667 + 0.0000i
  -3.0000 + 0.0000i
  -0.5000 + 0.0000i
  -2.0000 - 0.0000i
```

You can complete the round trip, which results in the expected price series, by the computation

```
pc = ret2tick(rc,Method="continuous",StartPrice=3)
```

```
pc =

   3.0000 + 0.0000i
   1.0000 + 0.0000i
  -2.0000 + 0.0000i
  -1.0000 + 0.0000i
   1.0000 + 0.0000i
```

Conclusion

Complex continuous returns are a necessary intermediary when considering logarithms of negative price ratios. `tick2ret` computes a continuous complex extension of the function on the positive real axis. The logarithm maintains the additivity property, used when computing multiperiod returns.

Because the extensible logarithm implemented in MATLAB, current implementations of Computational Finance tools that accept prices and returns behave logically with negative prices. The interpretation of complex-valued results can be unfamiliar at first, but as shown, the results are meaningful and explicable.

See Also

`tick2ret` | `ret2tick`

More About

- "Work with Negative Interest Rates Using Functions" (Financial Instruments Toolbox)

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:

- A collar is an 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.
- A hedge is a securities transaction that reduces or offsets the risk on an existing investment position.
- A straddle is 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.

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 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 Black-Scholes model was 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.

The binomial model, on the other hand, makes far fewer assumptions about the processes underlying an option. A binomial model is 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. 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 (where arbitrage is the purchase of securities on one market for immediate resale on another market to profit from a price or currency discrepancy).
- 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)
```

```
OptCall =

   13.6953


OptPut =

    6.3497


CallVal =

    0.6665


PutVal =

   -0.3335


GammaVal =

    0.0145


VegaVal =

   18.1843


LamCall =

    4.8664


LamPut =

   -5.2528
```

To summarize:

- 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)
```

```
Volatility =

    0.5000
```

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 =

  100.0000  111.2713  123.8732  137.9629  148.6915  166.2807
         0   89.9677  100.0495  111.3211  118.8981  132.9629
         0         0   80.9994   90.0175   95.0744  106.3211
         0         0         0   72.9825   76.0243   85.0175
         0         0         0         0   60.7913   67.9825
         0         0         0         0         0   54.3608
```

and the tree of option values.

```
OptionPrice =

   12.1011   19.1708   29.3470   42.9629   54.1653   71.2807
         0    5.3068    9.4081   16.3211   24.3719   37.9629
         0         0    1.3481    2.7402    5.5698   11.3211
         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

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

Related Examples

- “Handle and Convert Dates” on page 2-2
- “Greek-Neutral Portfolios of European Stock Options” on page 10-14
- “Plotting Sensitivities of an Option” on page 10-25
- “Plotting Sensitivities of a Portfolio of Options” on page 10-27

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 several characteristic features that are consequences of biological, cultural, and behavioral factors. Usually, 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.

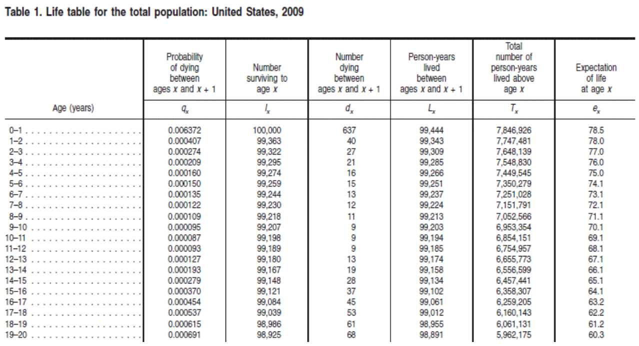


Table 1. Life table for the total population: United States, 2009

	Probability of dying between ages *x* and *x* + 1	Number surviving to age *x*	Number dying between ages *x* and *x* + 1	Person-years lived between ages *x* and *x* + 1	Total number of person-years lived above age *x*	Expectation of life at age *x*
Age (years)	q_x	l_x	d_x	L_x	T_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

Often, 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

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

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

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

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 2-46

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);
```

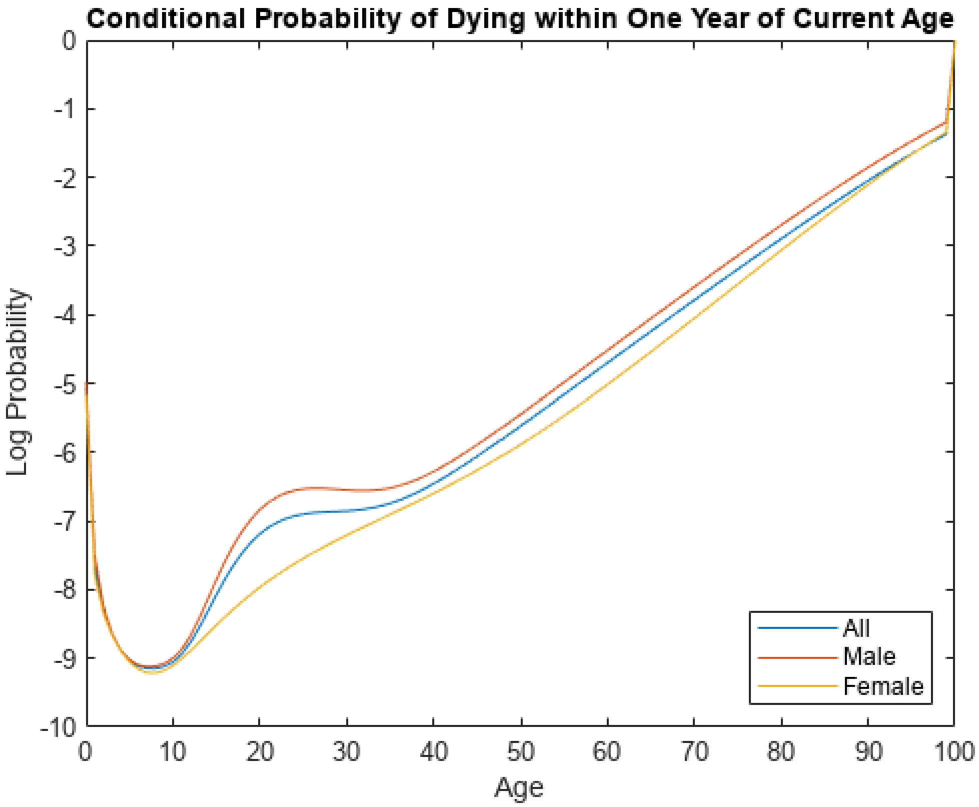
Generate life table series from the calibrated mortality model.

```
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');
```



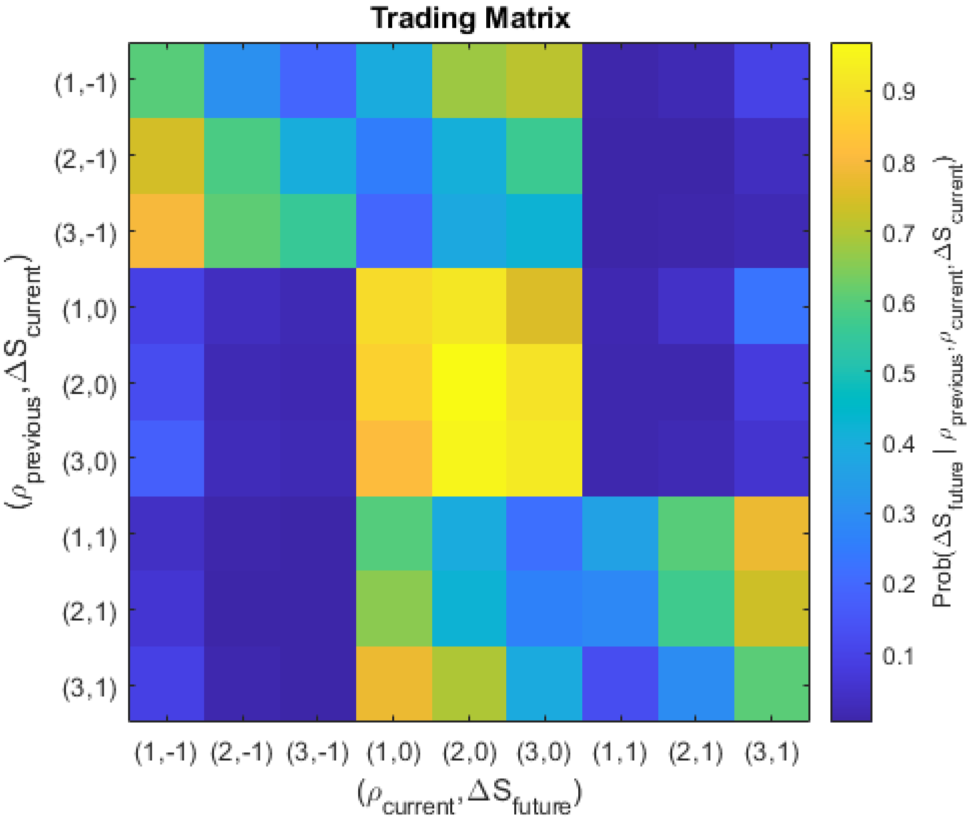
See Also

`lifetableconv` | `lifetablefit` | `lifetablegen`

More About

- “About Life Tables” on page 2-44

Machine Learning for Statistical Arbitrage: Introduction



Machine learning techniques for processing large amounts of data are broadly applicable in computational finance. The series of examples introduced in this topic provides a general workflow, illustrating how capabilities in MATLAB apply to a specific problem in financial engineering. The workflow is problem-oriented, exploratory, and guided by the data and the resulting analysis. The overall approach, however, is useful for constructing applications in many areas.

The workflow consists of these actions:

- Formulate a simple approach to algorithmic trading, through an analysis of market microstructure, with the goal of identifying real-time arbitrage opportunities.
- Use a large sample of exchange data to track order dynamics of a single security on a single day, selectively processing the data to develop relevant statistical measures.
- Create a model of intraday dynamics conditioned on a selection of hyperparameters introduced during feature engineering and development.
- Evaluate hyperparameter tunings using a supervising objective that computes cash returned on a model-based trading strategy.
- Optimize the trading strategy using different machine learning algorithms.
- Suggest modifications for further development.

The workflow is separated into three examples:

1. “Machine Learning for Statistical Arbitrage I: Data Management and Visualization” on page 2-50
2. “Machine Learning for Statistical Arbitrage II: Feature Engineering and Model Development” on page 2-60
3. “Machine Learning for Statistical Arbitrage III: Training, Tuning, and Prediction” on page 2-71

For more information about general workflows for machine learning, see:

- “Machine Learning in MATLAB”
- “Supervised Learning Workflow and Algorithms”

Machine Learning for Statistical Arbitrage I: Data Management and Visualization

This example shows techniques for managing, processing, and visualizing large amounts of financial data in MATLAB®. It is part of a series of related examples on machine learning for statistical arbitrage (see “Machine Learning Applications”).

Working with Big Data

Financial markets, with electronic exchanges such as NASDAQ executing orders on a timescale of milliseconds, generate vast amounts of data. Data streams can be mined for statistical arbitrage opportunities, but traditional methods for processing and storing dynamic analytic information can be overwhelmed by big data. Fortunately, new computational approaches have emerged, and MATLAB has an array of tools for implementing them.

Main computer memory provides high-speed access but limited capacity, whereas external storage offers low-speed access but potentially unlimited capacity. Computation takes place in memory. The computer recalls data and results from external storage.

Data Files

This example uses one trading day of NASDAQ exchange data [2] on one security (INTC) in a sample provided by LOBSTER [1] and included with Financial Toolbox™ in the zip file `LOBSTER_SampleFile_INTC_2012-06-21_5.zip`. Extract the contents of the zip file into your current folder. The expanded files, including two CSV files of data and the text file `LOBSTER_SampleFiles_ReadMe.txt`, consume 93.7 MB of memory.

```
unzip("LOBSTER_SampleFile_INTC_2012-06-21_5.zip");
```

The data describes the intraday evolution of the *limit order book* (LOB), which is the record of *market orders* (best price), *limit orders* (designated price), and resulting buys and sells. The data includes the precise time of these events, with orders tracked from arrival until cancellation or execution. At each moment in the trading day, orders on both the buy and sell side of the LOB exist at various *levels* away from the midprice between the lowest ask (order to sell) and the highest bid (order to buy).

Level 5 data (five levels away from the midprice on either side) is contained in two CSV files. Extract the trading date from the message file name.

```
MSGFileName = "INTC_2012-06-21_34200000_57600000_message_5.csv";   % Message file (description o
LOBFileName = "INTC_2012-06-21_34200000_57600000_orderbook_5.csv"; % Data file

[ticker,rem] = strtok(MSGFileName,'_');
date = strtok(rem,'_');
```

Data Storage

Daily data streams accumulate and need to be stored. A *datastore* is a repository for collections of data that are too big to fit in memory.

Use `tabularTextDatastore` to create datastores for the message and data files. Because the files contain data with different formats, create the datastores separately. Ignore generic column headers (for example, `VarName1`) by setting the `'ReadVariableNames'` name-value pair argument to `false`. Replace the headers with descriptive variable names obtained from

LOBSTER_SampleFiles_ReadMe.txt. Set the 'ReadSize' name-value pair argument to 'file' to allow similarly formatted files to be appended to existing datastores at the end of each trading day.

```
DSMSG = tabularTextDatastore(MSGFileName,'ReadVariableNames',false,'ReadSize','file');
DSMSG.VariableNames = ["Time","Type","OrderID","Size","Price","Direction"];

DSLOB = tabularTextDatastore(LOBFileName,'ReadVariableNames',false,'ReadSize','file');
DSLOB.VariableNames = ["AskPrice1","AskSize1","BidPrice1","BidSize1",...
                       "AskPrice2","AskSize2","BidPrice2","BidSize2",...
                       "AskPrice3","AskSize3","BidPrice3","BidSize3",...
                       "AskPrice4","AskSize4","BidPrice4","BidSize4",...
                       "AskPrice5","AskSize5","BidPrice5","BidSize5"];
```

Create a combined datastore by selecting Time and the level 3 data.

```
TimeVariable = "Time";
DSMSG.SelectedVariableNames = TimeVariable;

LOB3Variables = ["AskPrice1","AskSize1","BidPrice1","BidSize1",...
                 "AskPrice2","AskSize2","BidPrice2","BidSize2",...
                 "AskPrice3","AskSize3","BidPrice3","BidSize3"];
DSLOB.SelectedVariableNames = LOB3Variables;

DS = combine(DSMSG,DSLOB);
```

You can preview the first few rows in the combined datastore without loading data into memory.

```
DSPreview = preview(DS);
LOBPreview = DSPreview(:,1:5)
```

```
LOBPreview=8×5 table
    Time     AskPrice1    AskSize1    BidPrice1    BidSize1
    _____    _________    ________    _________    ________

    34200    2.752e+05       66       2.751e+05      400   
    34200    2.752e+05      166       2.751e+05      400   
    34200    2.752e+05      166       2.751e+05      400   
    34200    2.752e+05      166       2.751e+05      400   
    34200    2.752e+05      166       2.751e+05      300   
    34200    2.752e+05      166       2.751e+05      300   
    34200    2.752e+05      166       2.751e+05      300   
    34200    2.752e+05      166       2.751e+05      300   
```

The preview shows asks and bids *at the touch*, meaning the level 1 data, which is closest to the midprice. Time units are seconds after midnight, price units are dollar amounts times 10,000, and size units are the number of shares (see LOBSTER_SampleFiles_ReadMe.txt).

Tall Arrays and Timetables

Tall arrays work with out-of-memory data backed by a datastore using the MapReduce technique (see “Tall Arrays for Out-of-Memory Data”). When you use MapReduce, tall arrays remain unevaluated until you execute specific computations that use the data.

Set the execution environment for MapReduce to the local MATLAB session, instead of using Parallel Computing Toolbox™, by calling mapreducer(0). Then, create a tall array from the datastore DS by using tall. Preview the data in the tall array.

```
mapreducer(0)
DT = tall(DS);

DTPreview = DT(:,1:5)
```

```
DTPreview =

  Mx5 tall table

    Time     AskPrice1    AskSize1    BidPrice1    BidSize1
    _____    _________    ________    _________    ________

    34200    2.752e+05      66       2.751e+05      400
    34200    2.752e+05     166       2.751e+05      400
    34200    2.752e+05     166       2.751e+05      400
    34200    2.752e+05     166       2.751e+05      400
    34200    2.752e+05     166       2.751e+05      300
    34200    2.752e+05     166       2.751e+05      300
    34200    2.752e+05     166       2.751e+05      300
    34200    2.752e+05     166       2.751e+05      300
      :          :           :            :           :
      :          :           :            :           :
```

Timetables allow you to perform operations specific to time series (see “Create Timetables”). Because the LOB data consists of concurrent time series, convert DT to a tall timetable.

```
DT.Time = seconds(DT.Time); % Cast time as a duration from midnight.
DTT = table2timetable(DT);

DTTPreview = DTT(:,1:4)
```

```
DTTPreview =

  Mx4 tall timetable

      Time       AskPrice1    AskSize1    BidPrice1    BidSize1
    _________    _________    ________    _________    ________

    34200 sec    2.752e+05      66       2.751e+05      400
    34200 sec    2.752e+05     166       2.751e+05      400
    34200 sec    2.752e+05     166       2.751e+05      400
    34200 sec    2.752e+05     166       2.751e+05      400
    34200 sec    2.752e+05     166       2.751e+05      300
    34200 sec    2.752e+05     166       2.751e+05      300
    34200 sec    2.752e+05     166       2.751e+05      300
    34200 sec    2.752e+05     166       2.751e+05      300
        :            :           :            :           :
        :            :           :            :           :
```

Display all variables in the MATLAB workspace.

```
whos
```

```
  Name                Size            Bytes  Class                                       Attribute

  DS                  1x1                 8  matlab.io.datastore.CombinedDatastore
  DSLOB               1x1                 8  matlab.io.datastore.TabularTextDatastore
  DSMSG               1x1                 8  matlab.io.datastore.TabularTextDatastore
  DSPreview           8x13             4515  table
  DT                  Mx13               4950  tall
  DTPreview           Mx5                2840  tall
  DTT                 Mx12               4746  tall
  DTTPreview          Mx4                2650  tall
  LOB3Variables       1x12                936  string
  LOBFileName         1x1                 246  string
  LOBPreview          8x5                2203  table
  MSGFileName         1x1                 230  string
  TimeVariable        1x1                 150  string
  date                1x1                 166  string
  rem                 1x1                 230  string
  ticker              1x1                 150  string
```

Because all the data is in the datastore, the workspace uses little memory.

Preprocess and Evaluate Data

Tall arrays allow preprocessing, or *queuing*, of computations before they are evaluated, which improves memory management in the workspace.

Midprice S and imbalance index I are used to model LOB dynamics. To queue their computations, define them, and the time base, in terms of DTT.

```
timeBase = DTT.Time;
MidPrice = (DTT.BidPrice1 + DTT.AskPrice1)/2;

% LOB level 3 imbalance index:

lambda  = 0.5; % Hyperparameter
weights = exp(-(lambda)*[0 1 2]);
VAsk = weights(1)*DTT.AskSize1 + weights(2)*DTT.AskSize2 + weights(3)*DTT.AskSize3;
VBid = weights(1)*DTT.BidSize1 + weights(2)*DTT.BidSize2 + weights(3)*DTT.BidSize3;
ImbalanceIndex = (VBid-VAsk)./(VBid+VAsk);
```

The imbalance index is a weighted average of ask and bid volumes on either side of the midprice [3]. The imbalance index is a potential indicator of future price movements. The variable lambda is a *hyperparameter*, which is a parameter specified before training rather than estimated by the machine learning algorithm. A hyperparameter can influence the performance of the model. *Feature engineering* is the process of choosing domain-specific hyperparameters to use in machine learning algorithms. You can tune hyperparameters to optimize a trading strategy.

To bring preprocessed expressions into memory and evaluate them, use the gather function. This process is called *deferred evaluation*.

```
[t,S,I] = gather(timeBase,MidPrice,ImbalanceIndex);
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 1: Completed in 2.8 sec
Evaluation completed in 3 sec
```

A single call to gather evaluates multiple preprocessed expressions with a single pass through the datastore.

Determine the sample size, which is the number of *ticks*, or updates, in the data.

```
numTicks = length(t)
```

```
numTicks = 581030
```

The daily LOB data contains 581,030 ticks.

Checkpoint Data

You can save both unevaluated and evaluated data to external storage for later use.

Prepend the time base with the date, and cast the result as a datetime array. Save the resulting datetime array, MidPrice, and ImbalanceIndex to a MAT-file in a specified location.

```
dateTimeBase = datetime(date) + timeBase;
Today = timetable(dateTimeBase,MidPrice,ImbalanceIndex)
```

```
Today =

  581,030x2 tall timetable

        dateTimeBase          MidPrice     ImbalanceIndex
    ____________________    __________    ______________

    21-Jun-2012 09:30:00    2.7515e+05         -0.205    
    21-Jun-2012 09:30:00    2.7515e+05       -0.26006    
    21-Jun-2012 09:30:00    2.7515e+05       -0.26006    
    21-Jun-2012 09:30:00    2.7515e+05      -0.086772    
    21-Jun-2012 09:30:00    2.7515e+05       -0.15581    
    21-Jun-2012 09:30:00    2.7515e+05       -0.35382    
    21-Jun-2012 09:30:00    2.7515e+05       -0.19084    
    21-Jun-2012 09:30:00    2.7515e+05       -0.19084    
             :                  :               :
             :                  :               :
```

```
location = fullfile(pwd,"ExchangeData",ticker,date);
write(location,Today,'FileType','mat')
```

```
Writing tall data to folder C:\TEMP\Bdoc23a_2213998_3568\ib570499\19\tp1e1da1c0\finance-ex97702888
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 1: Completed in 3.5 sec
Evaluation completed in 3.8 sec
```

The file is written once, at the end of each trading day. The code saves the data to a file in a date-stamped folder. The series of ExchangeData subfolders serves as a historical data repository.

Alternatively, you can save workspace variables evaluated with gather directly to a MAT-file in the current folder.

```
save("LOBVars.mat","t","S","I")
```

In preparation for model validation later on, evaluate and add market order prices to the same file.

```
[MOBid,MOAsk] = gather(DTT.BidPrice1,DTT.AskPrice1);
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 1: Completed in 2.4 sec
Evaluation completed in 2.4 sec
```

```
save("LOBVars.mat","MOBid","MOAsk","-append")
```

The remainder of this example uses only the unevaluated tall timetable DTT. Clear other variables from the workspace.

```
clearvars -except DTT
whos
```

```
  Name            Size            Bytes  Class    Attributes

  DTT       581,030x12             4746  tall
```

Data Visualization

To visualize large amounts of data, you must summarize, bin, or sample the data in some way to reduce the number of points plotted on the screen.

LOB Snapshot

One method of visualization is to evaluate only a selected subsample of the data. Create a snapshot of the LOB at a specific time of day (11 AM).

```
sampleTimeTarget = seconds(11*60*60);               % Seconds after midnight
sampleTimes = withtol(sampleTimeTarget,seconds(1)); % 1 second tolerance
sampleLOB = DTT(sampleTimes,:);

numTimes = gather(size(sampleLOB,1))
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 1: Completed in 2.4 sec
Evaluation completed in 2.5 sec

numTimes = 23
```

There are 23 ticks within one second of 11 AM. For the snapshot, use the tick closest to the midtime.

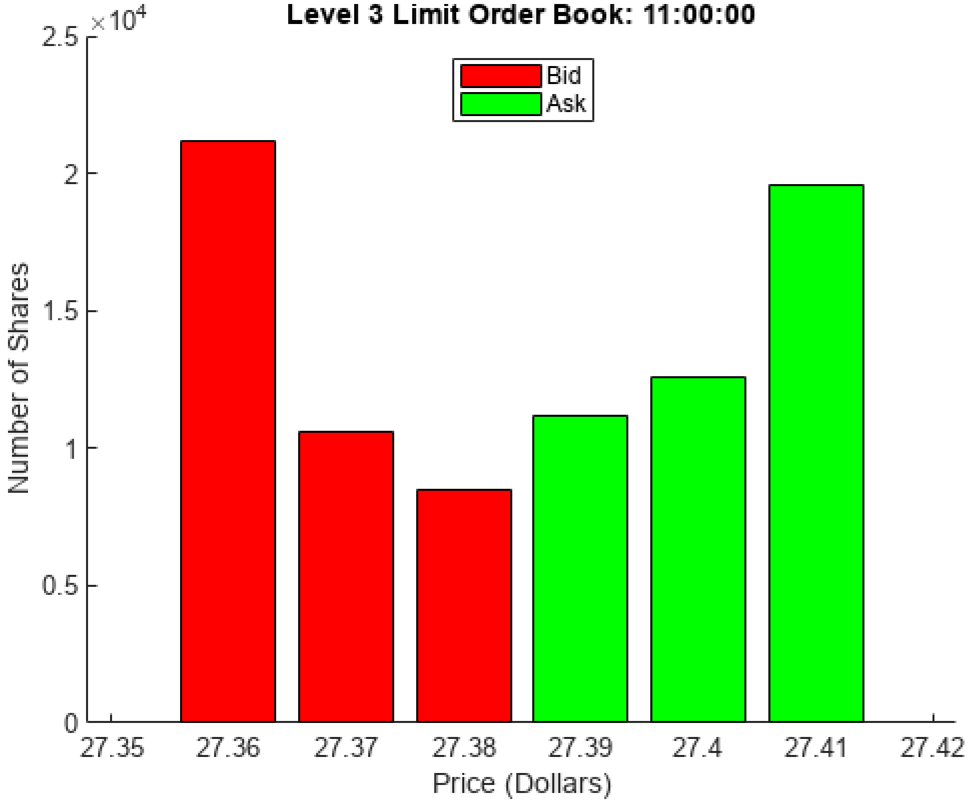
```
sampleLOB = sampleLOB(round(numTimes/2),:);
sampleTime = sampleLOB.Time;

sampleBidPrices = [sampleLOB.BidPrice1,sampleLOB.BidPrice2,sampleLOB.BidPrice3];
sampleBidSizes  = [sampleLOB.BidSize1,sampleLOB.BidSize2,sampleLOB.BidSize3];
sampleAskPrices = [sampleLOB.AskPrice1,sampleLOB.AskPrice2,sampleLOB.AskPrice3];
sampleAskSizes  = [sampleLOB.AskSize1,sampleLOB.AskSize2,sampleLOB.AskSize3];

[sampleTime,sampleBidPrices,sampleBidSizes,sampleAskPrices,sampleAskSizes] = ...
    gather(sampleTime,sampleBidPrices,sampleBidSizes,sampleAskPrices,sampleAskSizes);
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 2: Completed in 2.3 sec
- Pass 2 of 2: Completed in 2.6 sec
Evaluation completed in 5.4 sec
```

Visualize the limited data sample returned by `gather` by using `bar`.

```
figure
hold on

bar((sampleBidPrices/10000),sampleBidSizes,'r')
bar((sampleAskPrices/10000),sampleAskSizes,'g')
hold off

xlabel("Price (Dollars)")
ylabel("Number of Shares")
```

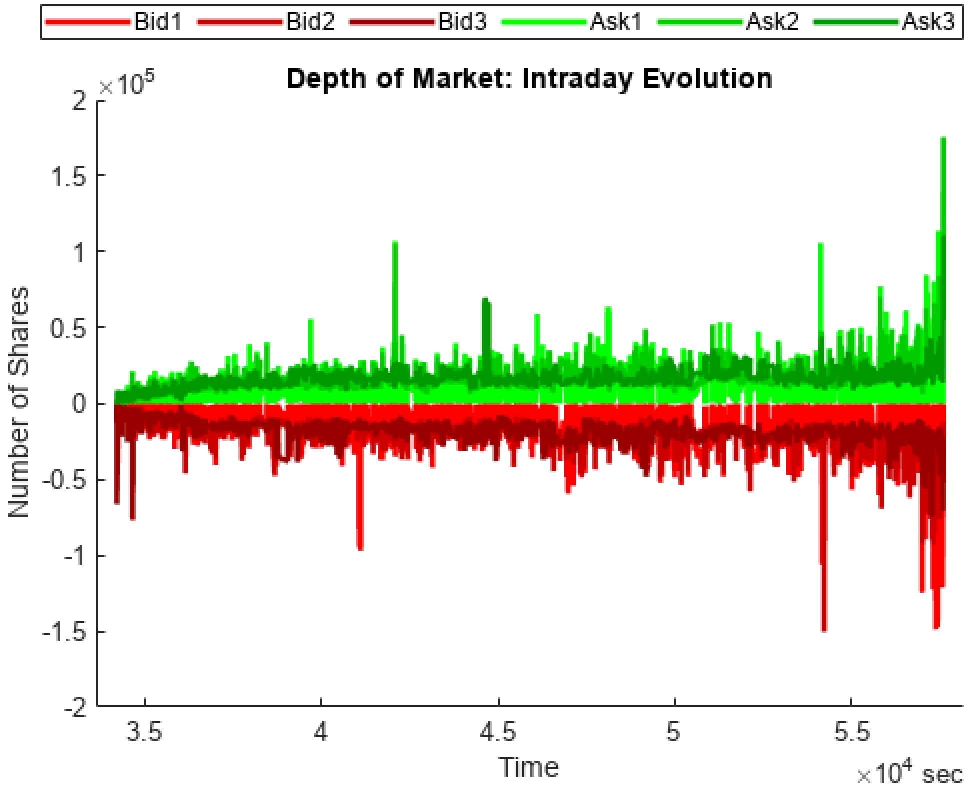
```
legend(["Bid","Ask"],'Location','North')
title(strcat("Level 3 Limit Order Book: ",datestr(sampleTime,"HH:MM:SS")))
```



Depth of Market

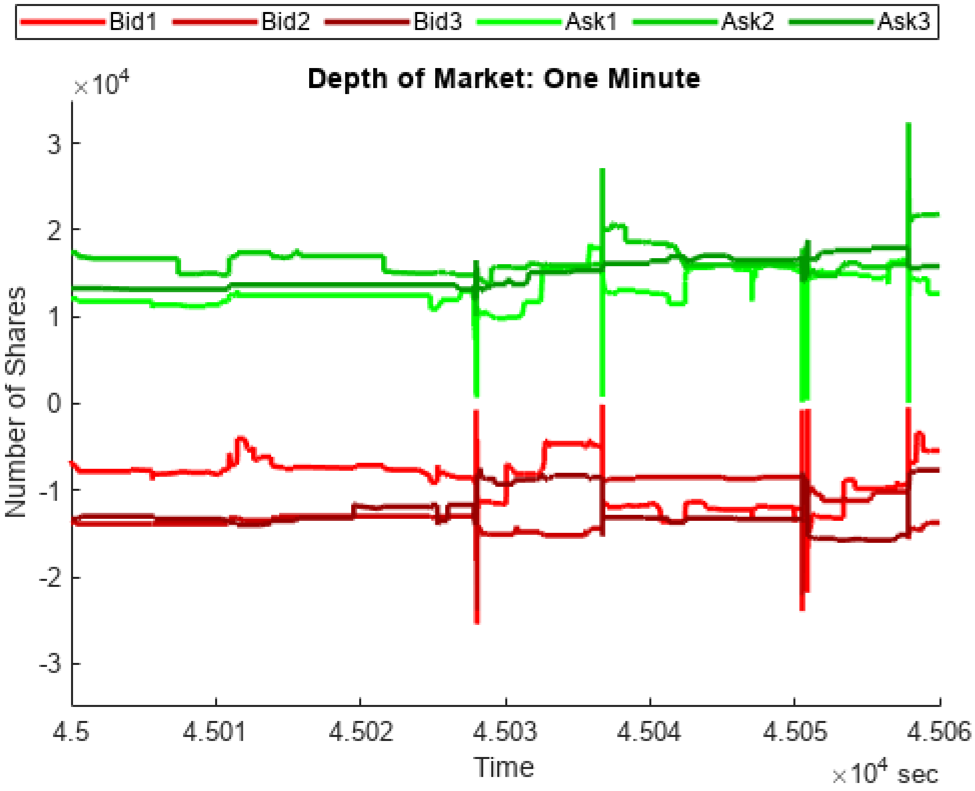
Some visualization functions work directly with tall arrays and do not require the use of `gather` (see “Visualization of Tall Arrays”). The functions automatically sample data to decrease pixel density. Visualize the level 3 intraday *depth of market,* which shows the time evolution of liquidity, by using `plot` with the tall timetable `DTT`.

```
figure
hold on

plot(DTT.Time,-DTT.BidSize1,'Color',[1.0 0 0],'LineWidth',2)
plot(DTT.Time,-DTT.BidSize2,'Color',[0.8 0 0],'LineWidth',2)
plot(DTT.Time,-DTT.BidSize3,'Color',[0.6 0 0],'LineWidth',2)

plot(DTT.Time,DTT.AskSize1,'Color',[0 1.0 0],'LineWidth',2)
plot(DTT.Time,DTT.AskSize2,'Color',[0 0.8 0],'LineWidth',2)
plot(DTT.Time,DTT.AskSize3,'Color',[0 0.6 0],'LineWidth',2)

hold off

xlabel("Time")
ylabel("Number of Shares")
title("Depth of Market: Intraday Evolution")
legend(["Bid1","Bid2","Bid3","Ask1","Ask2","Ask3"],'Location','NorthOutside','Orientation','Hori
```



To display details, limit the time interval.

```
xlim(seconds([45000 45060]))
ylim([-35000 35000])
title("Depth of Market: One Minute")
```



Summary

This example introduces the basics of working with big data, both in and out of memory. It shows how to set up, combine, and update external datastores, then create tall arrays for preprocessing data without allocating variables in the MATLAB workspace. The `gather` function transfers data into the workspace for computation and further analysis. The example shows how to visualize the data through data sampling or by MATLAB plotting functions that work directly with out-of-memory data.

References

[1] LOBSTER Limit Order Book Data. Berlin: frischedaten UG (haftungsbeschränkt).

[2] NASDAQ Historical TotalView-ITCH Data. New York: The Nasdaq, Inc.

[3] Rubisov, Anton D. "Statistical Arbitrage Using Limit Order Book Imbalance." Master's thesis, University of Toronto, 2015.

See Also

More About

- “Machine Learning for Statistical Arbitrage: Introduction” on page 2-48
- “Machine Learning for Statistical Arbitrage II: Feature Engineering and Model Development” on page 2-60

- “Machine Learning for Statistical Arbitrage III: Training, Tuning, and Prediction” on page 2-71

Machine Learning for Statistical Arbitrage II: Feature Engineering and Model Development

This example creates a continuous-time Markov model of limit order book (LOB) dynamics, and develops a strategy for algorithmic trading based on patterns observed in the data. It is part of a series of related examples on machine learning for statistical arbitrage (see “Machine Learning Applications”).

Exploratory Data Analysis

To predict the future behavior of a system, you need to discover patterns in historical data. The vast amount of data available from exchanges, such as NASDAQ, poses computational challenges while offering statistical opportunities. This example explores LOB data by looking for indicators of price momentum, following the approach in [4].

Raw Data

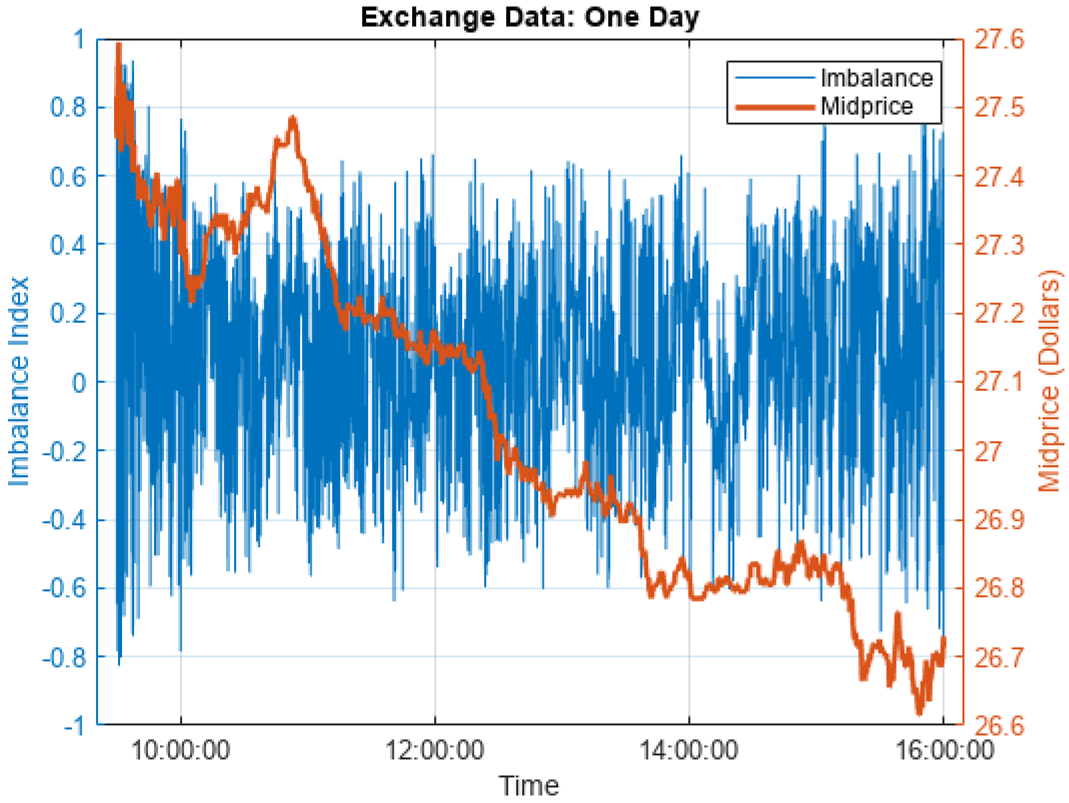
Load `LOBVars.mat`, the preprocessed LOB data set of the NASDAQ security INTC.

```
load LOBVars
```

The data set contains the following information for each order: the arrival time `t` (seconds from midnight), level 1 asking price `MOAsk`, level 1 bidding price `MOBid`, midprice `S`, and imbalance index `I`.

Create a plot that shows the intraday evolution of the LOB imbalance index `I` and midprice `S`.

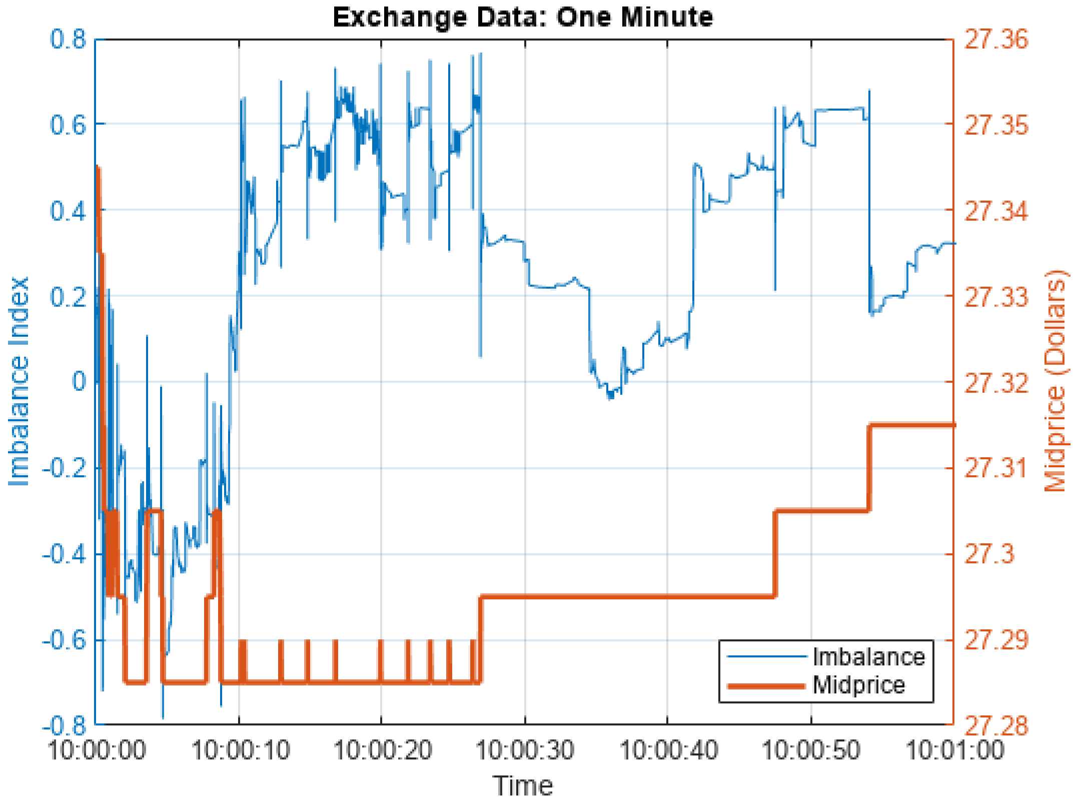
```
figure

t.Format = "hh:mm:ss";

yyaxis left
plot(t,I)
ylabel("Imbalance Index")

yyaxis right
plot(t,S/10000,'LineWidth',2)
ylabel("Midprice (Dollars)")

xlabel("Time")

title('Exchange Data: One Day')
legend(["Imbalance","Midprice"],'Location','NE')
grid on
```



At this scale, the imbalance index gives no indication of future changes in the midprice.

To see more detail, limit the time scale to one minute.

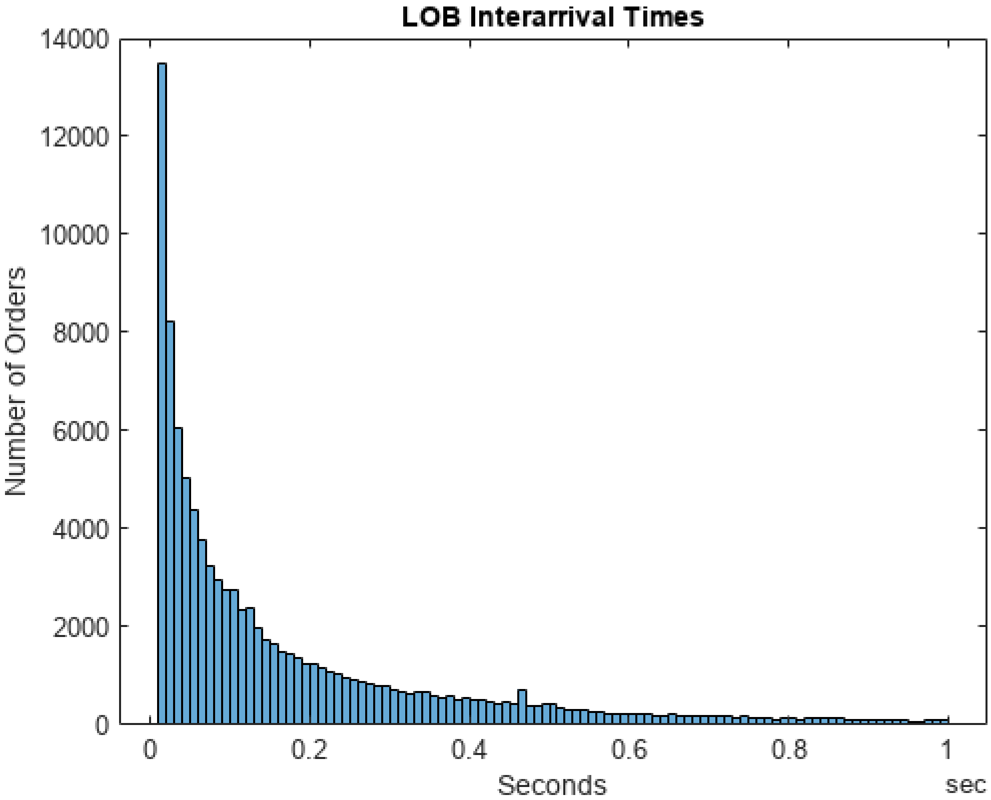
```
timeRange = seconds([36000 36060]); % One minute after 10 AM, when prices were climbing
xlim(timeRange)
legend('Location','SE')
title("Exchange Data: One Minute")
```



At this scale, sharp departures in the imbalance index align with corresponding departures in the midprice. If the relationship is predictive, meaning imbalances of a certain size forecast *future* price movements, then quantifying the relationship can provide statistical arbitrage opportunities.

Plot a histogram of the interarrival times in the LOB.

```
DT = diff(t); % Interarrival Times
DT.Format = "s";

figure
binEdges = seconds(0.01:0.01:1);
histogram(DT,binEdges)
xlabel("Seconds")
ylabel("Number of Orders")
title("LOB Interarrival Times")
```



Interarrival times follow the characteristic pattern of a Poisson process.

Compute the average wait time between orders by fitting an exponential distribution to the interarrival times.

```
DTAvg = expfit(DT)
```

```
DTAvg = duration
   0.040273 sec
```

Smoothed Data

The raw imbalance series I is erratic. To identify the most significant dynamic shifts, introduce a degree of smoothing dI, which is the number of backward ticks used to average the raw imbalance series.

```
dI = 10; % Hyperparameter
dTI = dI*DTAvg
```

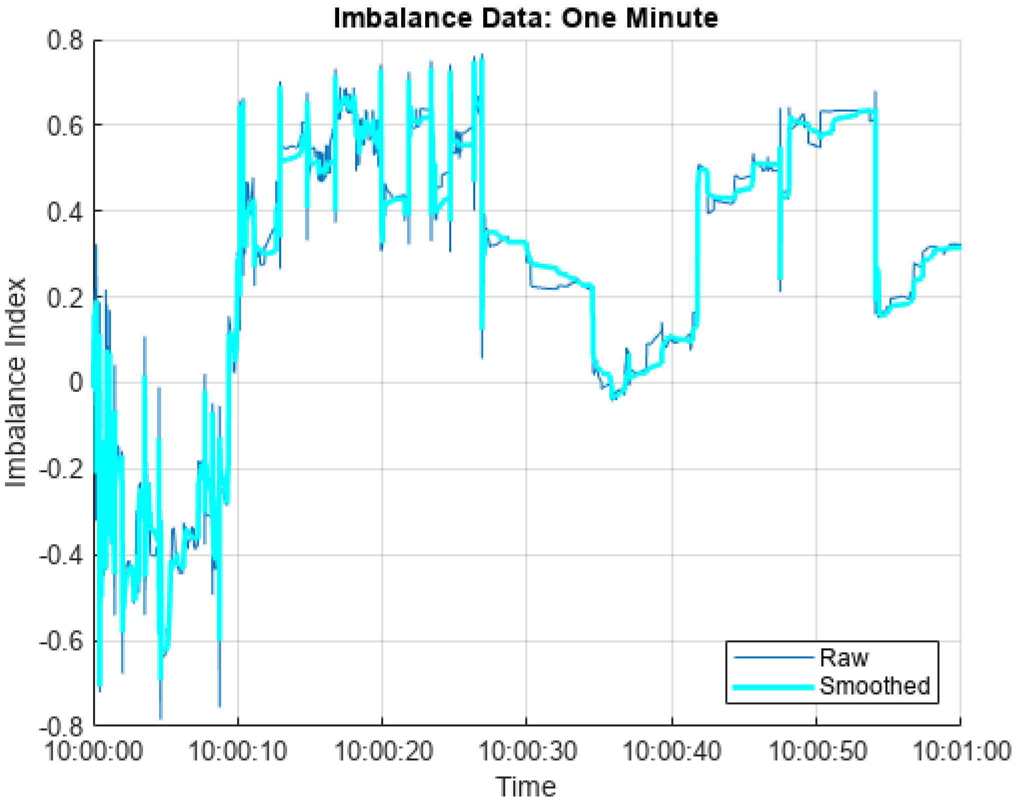
```
dTI = duration
   0.40273 sec
```

The setting corresponds to an interval of 10 ticks, or about 0.4 seconds on average. Smooth the imbalance indices over a trailing window.

```
sI = smoothdata(I,'movmean',[dI 0]);
```

Visualize the degree of smoothing to assess the volatility lost or retained.

```
figure
hold on
plot(t,I)
plot(t,sI,'c','LineWidth',2)
hold off

xlabel("Time")
xlim(timeRange)
ylabel("Imbalance Index")
title("Imbalance Data: One Minute")
legend(["Raw","Smoothed"],'Location','SE')
grid on
```



Discretized Data

To create a Markov model of the dynamics, collect the smoothed imbalance index `sI` into bins, discretizing it into a finite collection of states `rho` (ρ). The number of bins `numBins` is a hyperparameter.

```
numBins = 3; % Hyperparameter
binEdges = linspace(-1,1,numBins+1);
rho = discretize(sI,binEdges);
```

To model forecast performance, aggregate prices over a leading window. The number of ticks in a window `dS` is a hyperparameter.

```
dS = 20; % Hyperparameter
dTS = dS*DTAvg
```

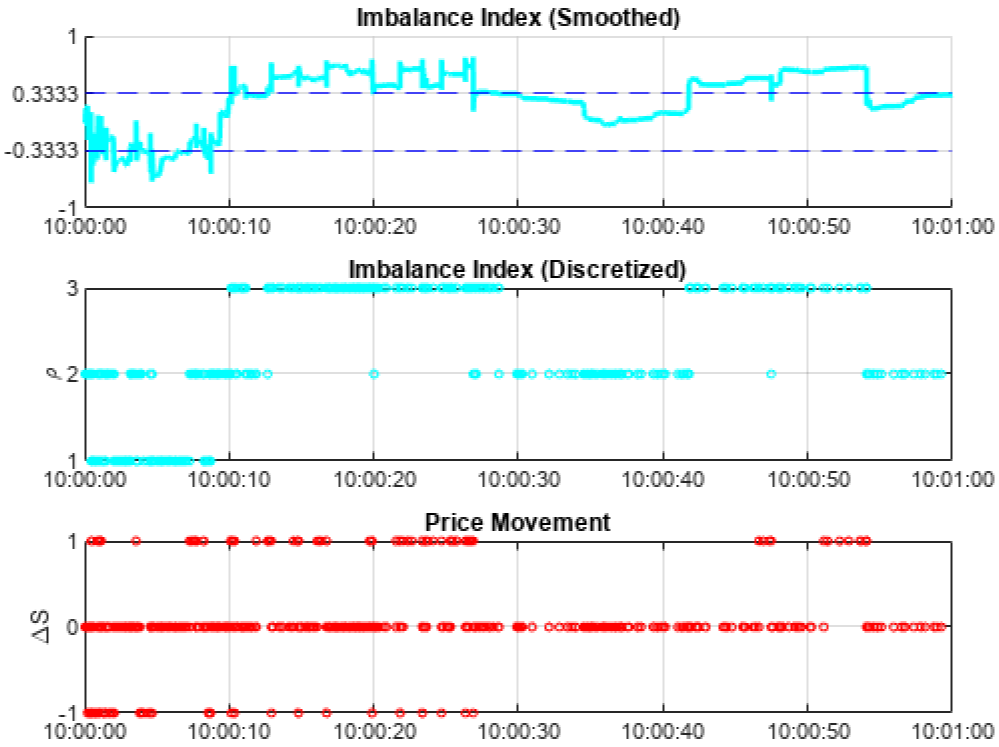
```
dTS = duration
   0.80547 sec
```

The setting corresponds to an interval of 20 ticks, or about 0.8 seconds on average. Discretize price movements into three states DS (ΔS) given by the sign of the forward price change.

```
DS = NaN(size(S));
shiftS = S(dS+1:end);
DS(1:end-dS) = sign(shiftS-S(1:end-dS));
```

Visualize the discretized data.

```
figure

subplot(3,1,1)
hold on
plot(t,sI,'c','LineWidth',2)
for i = 2:numBins
    yline(binEdges(i),'b--');
end
hold off
xlim(timeRange)
ylim([-1 1])
yticks(binEdges)
title("Imbalance Index (Smoothed)")
grid on

subplot(3,1,2)
plot(t,rho,'co','MarkerSize',3)
xlim(timeRange)
ylim([1 numBins])
yticks(1:numBins)
ylabel("\rho")
title("Imbalance Index (Discretized)")
grid on

subplot(3,1,3)
plot(t,DS,'ro','MarkerSize',3)
xlim(timeRange)
ylim([-1 1])
yticks([-1 0 1])
ylabel("\DeltaS")
title("Price Movement")
grid on
```



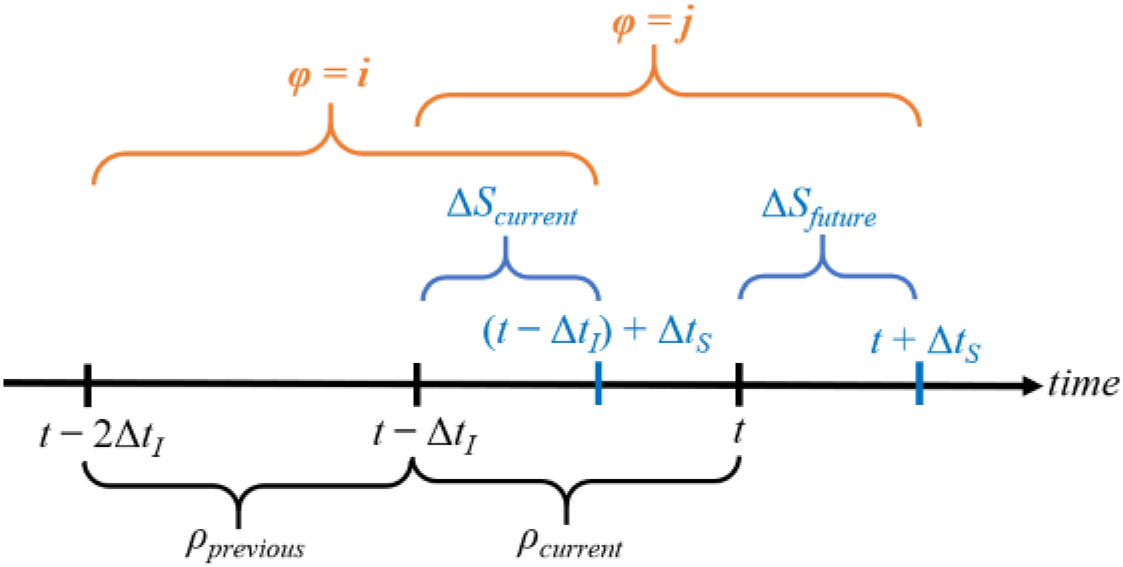
Continuous Time Markov Process

Together, the state of the LOB imbalance index `rho` (ρ) and the state of the forward price movement `DS` (ΔS) describe a two-dimensional continuous-time Markov chain (CTMC). The chain is modulated by the Poisson process of order arrivals, which signals any transition among the states.

To simplify the description, give the two-dimensional CTMC a one-dimensional encoding into states `phi` ($\varphi = (\rho, \Delta S)$).

```
numStates = 3*numBins; % numStates(DS)*numStates(rho)

phi = NaN(size(t));
for i = 1:length(t)
    switch DS(i)
        case -1
            phi(i) = rho(i);
        case 0
            phi(i) = rho(i) + numBins;
        case 1
            phi(i) = rho(i) + 2*numBins;
    end
end
```

Successive states of φ, and the component states ρ and ΔS, proceed as follows.



Hyperparameters dI (Δt_I) and dS (Δt_S) determine the size of a rolling state characterizing the dynamics. At time t, the process transitions from $\varphi = (\rho_{previous}, \Delta S_{current}) = i$ to $\varphi = (\rho_{current}, \Delta S_{future}) = j$ (or holds in the same state if $i = j$).

Estimate Process Parameters

Execution of the trading strategy at any time t is based on the probability of ΔS_{future} being in a particular state, conditional on the current and previous values of the other states. Following [3] and [4], determine empirical transition probabilities, and then assess them for predictive power.

```
% Transition counts

C = zeros(numStates);
for i = 1:length(phi)-dS-1
    C(phi(i),phi(i+1)) = C(phi(i),phi(i+1))+1;
end

% Holding times

H = diag(C);

% Transition rate matrix (infinitesimal generator)

G = C./H;
v = sum(G,2);
G = G + diag(-v);

% Transition probability matrix (stochastic for all dI)

P = expm(G*dI); % Matrix exponential
```

To obtain a trading matrix Q containing $\mathrm{Prob}(\Delta S_{future} \mid \rho_{previous}, \rho_{current}, \Delta S_{current})$ as in [4], apply Bayes' rule,

$$\mathrm{Prob}\left(\Delta S_{future}\,\middle|\,\rho_{previous},\rho_{current},\Delta S_{current}\right)=\frac{\mathrm{Prob}\left(\rho_{current},\Delta S_{future}\,\middle|\,\rho_{previous},\Delta S_{current}\right)}{\mathrm{Prob}\left(\rho_{current}\,\middle|\,\rho_{previous},\Delta S_{current}\right)}.$$

The numerator is the transition probability matrix `P`. Compute the denominator `PCond`.

```
PCond = zeros(size(P));
phiNums = 1:numStates;
modNums = mod(phiNums,numBins);
for i = phiNums
    for j = phiNums
        idx = (modNums == modNums(j));
        PCond(i,j) = sum(P(i,idx));
    end
end

Q = P./PCond;
```

Display `Q` in a table. Label the rows and columns with composite states $\varphi=(\rho,\Delta S)$.

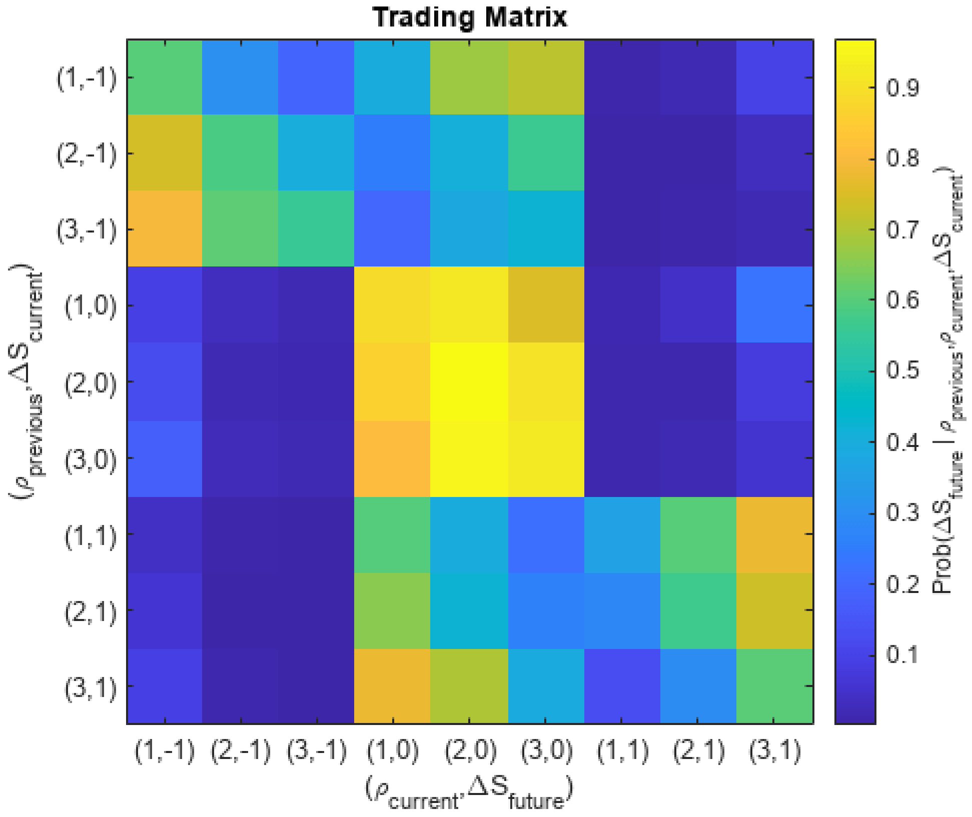
```
binNames = string(1:numBins);
stateNames = ["("+binNames+",-1)","("+binNames+",0)","("+binNames+",1)"];
QTable = array2table(Q,'RowNames',stateNames,'VariableNames',stateNames)
```

```
QTable=9×9 table
                (1,-1)      (2,-1)       (3,-1)       (1,0)      (2,0)      (3,0)       (1,1)
               ________    _________    _________    _______    _______    _______    _________

    (1,-1)      0.59952      0.30458      0.19165    0.39343    0.67723     0.7099    0.0070457
    (2,-1)      0.74092      0.58445      0.40023    0.25506    0.41003    0.56386    0.0040178
    (3,-1)      0.79895      0.60866      0.55443    0.19814      0.385    0.42501    0.0029096
    (1,0)      0.094173     0.036014     0.019107    0.88963    0.91688    0.75192     0.016195
    (2,0)       0.12325     0.017282     0.015453    0.86523    0.96939     0.9059     0.011525
    (3,0)        0.1773      0.02616     0.018494    0.81155    0.95359    0.92513     0.011154
    (1,1)      0.041132    0.0065127    0.0021313    0.59869    0.39374    0.21787      0.36017
    (2,1)      0.059151    0.0053554    0.0027769    0.65672    0.42325    0.26478      0.28413
    (3,1)      0.095832     0.010519    0.0051565     0.7768     0.6944     0.3906      0.12736
```

Rows are indexed by $(\rho_{previous},\Delta S_{current})$. Conditional probabilities for each of the three possible states of ΔS_{future} are read from the corresponding column, conditional on $\rho_{current}$.

Represent `Q` with a heatmap.

```
figure
imagesc(Q)
axis equal tight
hCB = colorbar;
hCB.Label.String = "Prob(\DeltaS_{future} | \rho_{previous},\rho_{current},\DeltaS_{current})";
xticks(phiNums)
xticklabels(stateNames)
xlabel("(\rho_{current},\DeltaS_{future})")
yticks(phiNums)
yticklabels(stateNames)
ylabel("(\rho_{previous},\DeltaS_{current})")
title("Trading Matrix")
```



The bright, central 3 x 3 square shows that in most transitions, tick to tick, no price change is expected ($\Delta S_{future} = 0$). Bright areas in the upper-left 3 x 3 square (downward price movements $\Delta S_{future} = -1$) and lower-right 3 x 3 square (upward price movements $\Delta S_{future} = +1$) show evidence of momentum, which can be leveraged in a trading strategy.

You can find arbitrage opportunities by thresholding `Q` above a specified trigger probability. For example:

```
trigger = 0.5;
QPattern = (Q > trigger)
```

```
QPattern = 9x9 logical array

   1   0   0   0   1   1   0   0   0
   1   1   0   0   0   1   0   0   0
   1   1   1   0   0   0   0   0   0
   0   0   0   1   1   1   0   0   0
   0   0   0   1   1   1   0   0   0
   0   0   0   1   1   1   0   0   0
   0   0   0   1   0   0   0   1   1
   0   0   0   1   0   0   0   1   1
   0   0   0   1   1   0   0   0   1
```

The entry in the (1,1) position shows a chance of more than 50% that a downward price movement ($\Delta S_{current} = -1$) will be followed by another downward price movement ($\Delta S_{future} = -1$), provided that the previous and current imbalance states ρ are both 1.

A Trading Strategy?

Q is constructed on the basis of both the available exchange data and the hyperparameter settings. Using Q to inform future trading decisions depends on the market continuing in the same statistical pattern. Whether the market exhibits momentum in certain states is a test of the weak-form *Efficient Market Hypothesis* (EMH). For heavily traded assets, such as the one used in this example (INTC), the EMH is likely to hold over extended periods, and arbitrage opportunities quickly disappear. However, failure of EMH can occur in some assets over short time intervals. A working trading strategy divides a portion of the trading day, short enough to exhibit a degree of statistical equilibrium, into a training period for estimating Q, using optimal hyperparameter settings and a validation period on which to trade. For an implementation of such a strategy, see “Machine Learning for Statistical Arbitrage III: Training, Tuning, and Prediction” on page 2-71.

Summary

This example begins with raw data on the LOB and transforms it into a summary (the Q matrix) of statistical arbitrage opportunities. The analysis uses the mathematics of continuous-time Markov chain models, first in recognizing the Poisson process of LOB interarrival times, then by discretizing data into two-dimensional states representing the instantaneous position of the market. A description of state transitions, derived empirically, leads to the possibility of an algorithmic trading strategy.

References

[1] Cartea, Álvaro, Sebastian Jaimungal, and Jason Ricci. "Buy Low, Sell High: A High-Frequency Trading Perspective." *SIAM Journal on Financial Mathematics* 5, no. 1 (January 2014): 415–44. https://doi.org/10.1137/130911196.

[2] Guilbaud, Fabien, and Huyen Pham. "Optimal High-Frequency Trading with Limit and Market Orders." *Quantitative Finance* 13, no. 1 (January 2013): 79–94. https://doi.org/10.1080/14697688.2012.708779.

[3] Norris, J. R. *Markov Chains*. Cambridge, UK: Cambridge University Press, 1997.

[4] Rubisov, Anton D. "Statistical Arbitrage Using Limit Order Book Imbalance." Master's thesis, University of Toronto, 2015.

See Also

More About

- “Machine Learning for Statistical Arbitrage: Introduction” on page 2-48
- “Machine Learning for Statistical Arbitrage I: Data Management and Visualization” on page 2-50
- “Machine Learning for Statistical Arbitrage III: Training, Tuning, and Prediction” on page 2-71

Machine Learning for Statistical Arbitrage III: Training, Tuning, and Prediction

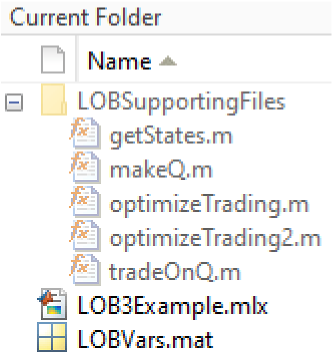
This example uses Bayesian optimization to tune hyperparameters in the algorithmic trading model, supervised by the end-of-day return. It is part of a series of related examples on machine learning for statistical arbitrage (see “Machine Learning Applications”).

Load `LOBVars.mat`, the preprocessed LOB data set of the NASDAQ security INTC.

```
load LOBVars
```

The data set contains the following information for each order: the arrival time `t` (seconds from midnight), level 1 asking price `MOAsk`, level 1 bidding price `MOBid`, midprice `S`, and imbalance index `I`.

This example includes several supporting functions. To view them, open the example, and then expand the `LOBSupportingFiles` folder in the **Current Folder** pane.



Current Folder

Name

LOBSupportingFiles
- getStates.m
- makeQ.m
- optimizeTrading.m
- optimizeTrading2.m
- tradeOnQ.m

LOB3Example.mlx

LOBVars.mat

Access the files by adding them to the search path.

```
addpath("LOBSupportingFiles")
```

Trading Strategy

The trading matrix `Q` contains probabilities of future price movements, given current and previous states `rho` of the limit order book (LOB) imbalance index `I` and the latest observed direction in prices `DS`.

View the supporting function `tradeOnQ.m`, which implements a simple trading strategy based on the pattern in `Q`.

```
function cash = tradeOnQ(Data,Q,n,N)

% Reference: Machine Learning for Statistical Arbitrage
%            Part II: Feature Engineering and Model Development

% Data

t = Data.t;
MOBid = Data.MOBid;
MOAsk = Data.MOAsk;
```

```
% States

[rho,DS] = getStates(Data,n,N);

% Start of trading

cash = 0;
assets = 0;

% Active trading

T = length(t);

for tt = 2:T-N  % Trading ticks

    % Get Q row, column indices of current state

    row = rho(tt-1)+n*(DS(tt-1)+1);
    downColumn = rho(tt);
    upColumn = rho(tt) + 2*n;

    % If predicting downward price move

    if Q(row,downColumn) > 0.5

        cash = cash + MOBid(tt); % Sell
        assets = assets - 1;

    % If predicting upward price move

    elseif Q(row,upColumn) > 0.5

        cash = cash - MOAsk(tt); % Buy
        assets = assets + 1;

    end

end

% End of trading (liquidate position)

if assets > 0

    cash = cash + assets*MOBid(T); % Sell off

elseif assets < 0

    cash = cash + assets*MOAsk(T); % Buy back

end
```

The algorithm uses predictions from `Q` to make decisions about trading at each tick. It illustrates the general mechanism of any optimized machine learning algorithm.

This strategy seeks to profit from expected price changes using market orders (best offer at the touch) of a single share at each tick, if an arbitrage opportunity arises. The strategy can be scaled up to larger trading volumes. Using the conditional probabilities obtained from Q, the tradeOnQ function takes one of these actions:

- Executes a buy if the probability of an upward forward price change is greater than 0.5.
- Executes a sell if the probability of an downward forward price change is greater than 0.5.

At the end of the trading day, the function liquidates the position at the touch.

The strategy requires Data with tick times t and the corresponding market order bid and ask prices MOBid and MOAsk, respectively. In real-time trading, data is provided by the exchange. This example evaluates the strategy by dividing the historical sample into *training* (calibration) and *validation* subsamples. The validation subsample serves as a proxy for real-time trading data. The strategy depends on Q, the trading matrix itself, which you estimate after you make a number of hyperparameter choices. The inputs n and N are hyperparameters to tune when you optimize the strategy.

Hyperparameters

The continuous-time Markov model and the resulting trading matrix Q depend on the values of four hyperparameters:

- lambda — The weighting parameter used to compute the imbalance index I
- dI — The number of backward ticks used to average I during smoothing
- numBins — The number of bins used to partition smoothed I for discretization
- dS — The number of forward ticks used to convert the prices S to discrete DS

In general, all four hyperparameters are tunable. However, to facilitate visualization, the example reduces the number of dimensions by tuning only numBins and N. The example:

- Fixes lambda
- Leaves numBins = n, where n is free to vary
- Equalizes the window lengths dI = dS = N, where N is free to vary

The restrictions do not significantly affect optimization outcomes. The optimization algorithm searches over the two-dimensional parameter space (n,N) for the configuration yielding the maximum return on trading.

Training and Validation Data

Machine learning requires a subsample on which to estimate Q and another subsample on which to evaluate the hyperparameter selections.

Specify a breakpoint to separate the data into training and validation subsamples. The breakpoint affects evaluation of the objective function, and is essentially another hyperparameter. However, because you do not tune the breakpoint, it is external to the optimization process.

```
bp = round((0.80)*length(t)); % Use 80% of data for training
```

Collect data in a timetable to pass to tradeOnQ.

```
Data = timetable(t,S,I,MOBid,MOAsk);
TData = Data(1:bp,:);       % Training data
VData = Data(bp+1:end,:);   % Validation data
```

Cross-Validation

Cross-validation describes a variety of techniques to assess how training results (here, computation of Q) generalize, with predictive reliability, to independent validation data (here, profitable trading). The goal of cross-validation is to flag problems in training results, like bias and overfitting. In the context of the trading strategy, overfitting refers to the time dependence, or nonstationarity, of Q. As Q changes over time, it becomes less effective in predicting future price movements. The key diagnostic issue is the degree to which Q changes, and at what rate, over a limited trading horizon.

With training and validation data in place, specify the hyperparameters and compare Q in the two subsamples. The supporting function makeQ.m provides the steps for making Q.

```
% Set specific hyperparameters

n = 3;  % Number of bins for I
N = 20; % Window lengths

% Compare Qs

QT = makeQ(TData,n,N);
QV = makeQ(VData,n,N);
QTVDiff = QT - QV
```

```
QTVDiff = 9×9

    0.0070    0.0182    0.1198   -0.0103   -0.0175   -0.0348    0.0034   -0.0007   -0.0851
   -0.0009    0.0176    0.2535   -0.0010   -0.0233   -0.2430    0.0019    0.0058   -0.0106
    0.0184    0.0948    0.0835   -0.0195   -0.1021   -0.1004    0.0011    0.0073    0.0168
    0.0462    0.0180    0.0254   -0.0512   -0.0172    0.0417    0.0050   -0.0009   -0.0671
    0.0543    0.0089    0.0219   -0.0556   -0.0169   -0.0331    0.0013    0.0080    0.0112
    0.1037    0.0221    0.0184   -0.1043   -0.0401   -0.0479    0.0006    0.0180    0.0295
    0.0266    0.0066    0.0054   -0.0821   -0.0143   -0.0116    0.0555    0.0077    0.0062
    0.0615    0.0050    0.0060   -0.0189   -0.0207   -0.0262   -0.0426    0.0157    0.0203
    0.0735    0.0103    0.0090   -0.0788   -0.1216   -0.0453    0.0053    0.1113    0.0362
```

Differences between QT and QV appear minor, although they vary based on their position in the matrix. Identify trading inefficiencies, which result from indices (market states) where one matrix gives a trading cue (probability value > 0.5) and the other does not.

```
Inhomogeneity = (QT > 0.5 & QV < 0.5 ) | (QT < 0.5 & QV > 0.5 )
```

```
Inhomogeneity = 9x9 logical array

   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   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
```

No significant inhomogeneities appear in the data with the given hyperparameter settings.

The severity of proceeding with a homogeneity assumption is not known *a priori*, and can emerge only from more comprehensive backtesting. Statistical tests are available, as described in [4] and [5], for example. During real-time trading, a rolling computation of `Q` over trailing training data of suitable size can provide the most reliable cues. Such an approach acknowledges inherent nonstationarity in the market.

Machine Learning

Machine learning refers to the general approach of effectively performing a task (for example, trading) in an automated fashion by detecting patterns (for example, computing `Q`) and making inferences based on available data. Often, data is dynamic and big enough to require specialized computational techniques. The evaluation process—tuning hyperparameters to describe the data and direct performance of the task—is ongoing.

In addition to the challenges of working with big data, the process of evaluating complex, sometimes black-box, objective functions is also challenging. Objective functions supervise hyperparameter evaluation. The trading strategy evaluates hyperparameter tunings by first computing `Q` on a training subsample, and then trading during an evaluation (real-time) subsample. The objective is to maximize profit, or minimize negative cash returned, over a space of suitably constrained configurations (`n`,`N`). This objective is a prototypical "expensive" objective function. *Bayesian optimization* is a type of machine learning suited to such objective functions. One of its principle advantages is the absence of costly derivative evaluations. To implement Bayesian optimization, use the Statistics and Machine Learning Toolbox™ function `bayesopt`.

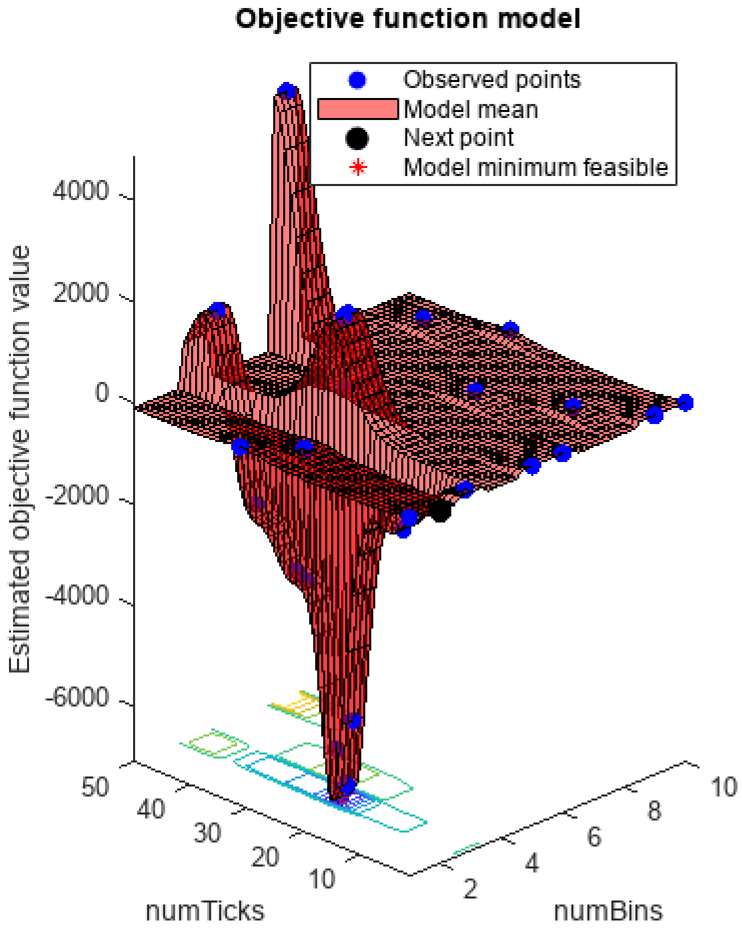
The supporting function `optimizeTrading.m` uses `bayesopt` to optimize the trading strategy in `tradeOnQ`.

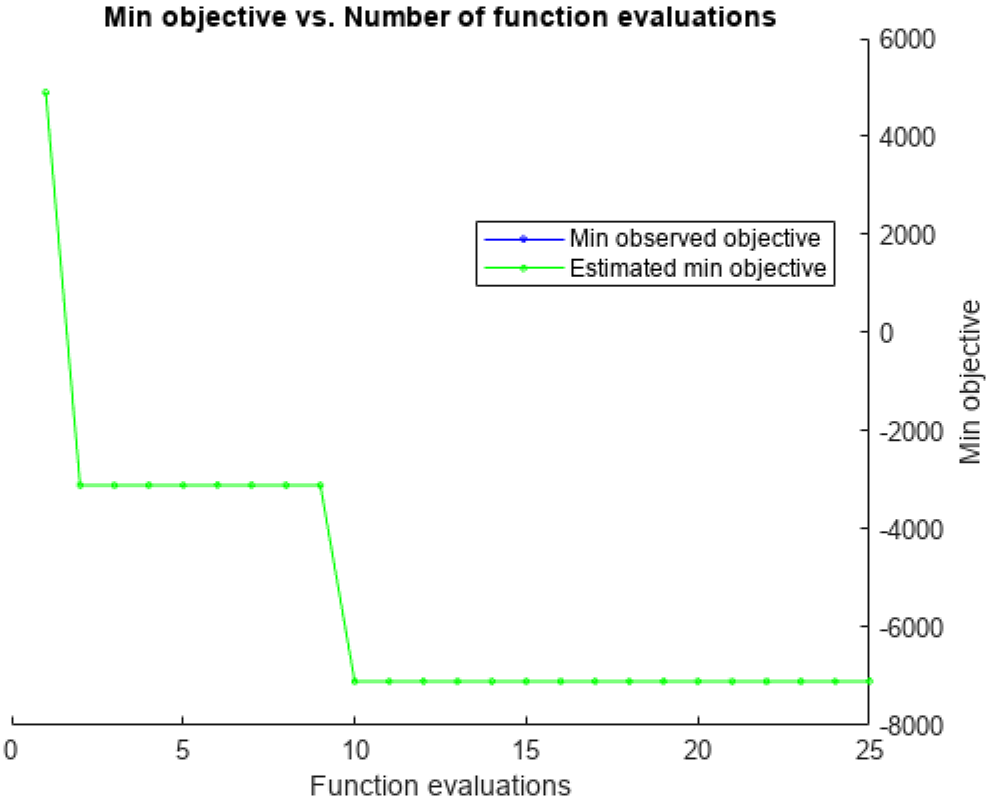
```
function results = optimizeTrading(TData,VData)

% Optimization variables

n = optimizableVariable('numBins',[1 10],'Type','integer');
N = optimizableVariable('numTicks',[1 50],'Type','integer');

% Objective function handle

f = @(x)negativeCash(x,TData,VData);

% Optimize

results = bayesopt(f,[n,N],...
                   'IsObjectiveDeterministic',true,...
                   'AcquisitionFunctionName','expected-improvement-plus',...
                   'MaxObjectiveEvaluations',25,...
                   'ExplorationRatio',2,...
                   'Verbose',0);

end % optimizeTrading

% Objective (local)
function loss = negativeCash(x,TData,VData)

n = x.numBins;
N = x.numTicks;

% Make trading matrix Q
```

```
Q = makeQ(TData,n,N);

% Trade on Q

cash = tradeOnQ(VData,Q,n,N);

% Objective value

loss = -cash;

end % negativeCash
```

Optimize the trading strategy by passing the training and validation data to `optimizeTrading`.

```
rng(0) % For reproducibility
results = optimizeTrading(TData,VData);
```





The estimated minimum objective coincides with the minimum observed objective (the search is monotonic). Unlike derivative-based algorithms, `bayesopt` does not converge. As it tries to find the global minimum, `bayesopt` continues exploring until it reaches the specified number of iterations (25).

Obtain the best configuration by passing the `results` to `bestPoint`.

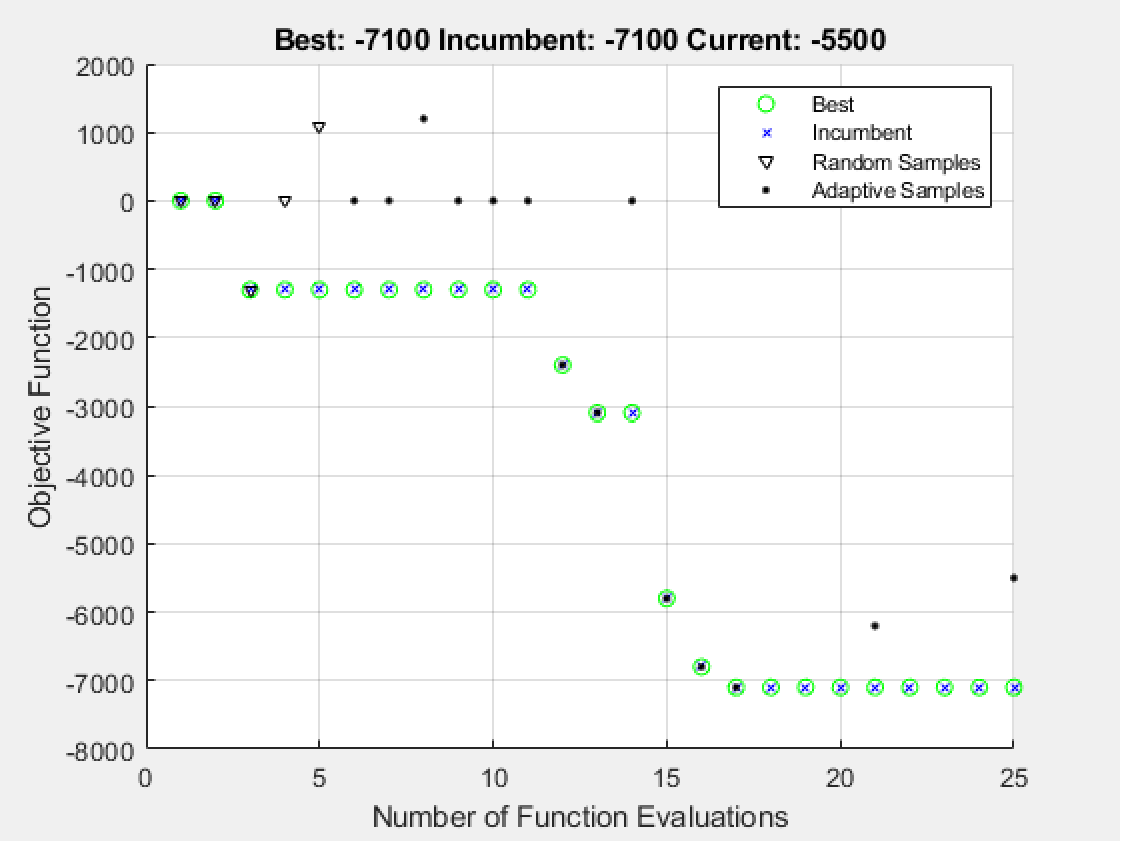
```
[Calibration,negReturn] = bestPoint(results,'Criterion','min-observed')
```

```
Calibration=1×2 table
    numBins    numTicks
    _______    ________

       3          24   

negReturn = -7100
```

Trading one share per tick, as directed by `Q`, the optimal strategy using (n,N) = (3,24) returns $0.71 over the final 20% of the trading day. Modifying the trading volume scales the return.

Another optimizer designed for expensive objectives is `surrogateopt` (Global Optimization Toolbox). It uses a different search strategy and can locate optima more quickly, depending on the objective. The supporting function `optimizeTrading2.m` uses `surrogateopt` instead of `bayesopt` to optimize the trading strategy in `tradeOnQ`.

```
rng(0) % For reproducibility
results2 = optimizeTrading2(TData,VData)
```



```
results2 = 1×2

     3    24
```

The results obtained with `surrogateopt` are the same as the `bayesopt` results. The plot contains information about the progress of the search that is specific to the `surrogateopt` algorithm.

Compute `Q` by passing the optimal hyperparameters and the entire data set to `makeQ`.

```
bestQ = makeQ(Data,3,24)
```

```
bestQ = 9×9

    0.3933    0.1868    0.1268    0.5887    0.7722    0.6665    0.0180    0.0410    0.2068
    0.5430    0.3490    0.2716    0.4447    0.6379    0.6518    0.0123    0.0131    0.0766
    0.6197    0.3897    0.3090    0.3705    0.5954    0.6363    0.0098    0.0150    0.0547
    0.1509    0.0440    0.0261    0.8217    0.8960    0.6908    0.0273    0.0601    0.2831
    0.1900    0.0328    0.0280    0.7862    0.9415    0.8316    0.0238    0.0257    0.1404
    0.2370    0.0441    0.0329    0.7391    0.9221    0.8745    0.0239    0.0338    0.0925
    0.1306    0.0234    0.0101    0.7861    0.6566    0.4168    0.0833    0.3200    0.5731
    0.1276    0.0169    0.0118    0.7242    0.6505    0.4712    0.1482    0.3326    0.5171
    0.1766    0.0282    0.0186    0.7216    0.7696    0.6185    0.1018    0.2023    0.3629
```

The trading matrix `bestQ` can be used as a starting point for the next trading day.

Summary

This example implements the optimized trading strategy developed in the first two related examples. Available data is split into training and validation subsamples and used, respectively, to compute the trading matrix `Q` and execute the resulting trading algorithm. The process is repeated over a space of hyperparameter settings using the global optimizers `bayesopt` and `surrogateopt`, both of which identify an optimal strategy yielding a positive return. The approach has many options for further customization.

References

[1] Bull, Adam D. "Convergence Rates of Efficient Global Optimization Algorithms." *Journal of Machine Learning Research* 12, (November 2011): 2879–904.

[2] Rubisov, Anton D. "Statistical Arbitrage Using Limit Order Book Imbalance." Master's thesis, University of Toronto, 2015.

[3] Snoek, Jasper, Hugo Larochelle, and Ryan P. Adams. "Practical Bayesian Optimization of Machine Learning Algorithms." In *Advances in Neural Information Processing Systems 25*, F. Pereira et. al. editors, 2012.

[4] Tan, Barış, and Kamil Yılmaz. “Markov Chain Test for Time Dependence and Homogeneity: An Analytical and Empirical Evaluation.” *European Journal of Operational Research* 137, no. 3 (March 2002): 524–43. https://doi.org/10.1016/S0377-2217(01)00081-9.

[5] Weißbach, Rafael, and Ronja Walter. “A Likelihood Ratio Test for Stationarity of Rating Transitions.” *Journal of Econometrics* 155, no. 2 (April 2010): 188–94. https://doi.org/10.1016/j.jeconom.2009.10.016.

See Also

More About

- “Machine Learning for Statistical Arbitrage: Introduction” on page 2-48
- “Machine Learning for Statistical Arbitrage I: Data Management and Visualization” on page 2-50
- “Machine Learning for Statistical Arbitrage II: Feature Engineering and Model Development” on page 2-60

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

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

Related Examples

- “Portfolio Optimization Functions” on page 3-3
- “Portfolio Construction Examples” on page 3-5
- “Portfolio Selection and Risk Aversion” on page 3-7
- “Active Returns and Tracking Error Efficient Frontier” on page 3-25
- “Plotting an Efficient Frontier Using portopt” on page 10-22

More About

- “Portfolio Object Workflow” on page 4-17

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
`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 3-11.

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 4-17.

See Also

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

Related Examples

- “Portfolio Construction Examples” on page 3-5
- “Portfolio Selection and Risk Aversion” on page 3-7
- “Active Returns and Tracking Error Efficient Frontier” on page 3-25
- “Plotting an Efficient Frontier Using portopt” on page 10-22
- “portopt Migration to Portfolio Object” on page 3-11

More About

- “Analyzing Portfolios” on page 3-2
- “Portfolio Object Workflow” on page 4-17

Portfolio Construction Examples

In this section...
"Introduction" on page 3-5
"Efficient Frontier Example" on page 3-5

Introduction

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

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 4-17.

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 4-95
- "Obtaining Endpoints of the Efficient Frontier" on page 4-98
- "Obtaining Efficient Portfolios for Target Returns" on page 4-101
- "Obtaining Efficient Portfolios for Target Risks" on page 4-104
- "Efficient Portfolio That Maximizes Sharpe Ratio" on page 4-107
- "Estimate Efficient Frontiers for Portfolio Object" on page 4-118
- "Plotting the Efficient Frontier for a Portfolio Object" on page 4-121

See Also

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

Related Examples

- "Portfolio Optimization Functions" on page 3-3
- "Portfolio Selection and Risk Aversion" on page 3-7
- "Active Returns and Tracking Error Efficient Frontier" on page 3-25
- "Plotting an Efficient Frontier Using portopt" on page 10-22
- "portopt Migration to Portfolio Object" on page 3-11

More About

- “Analyzing Portfolios” on page 3-2
- “Portfolio Object Workflow” on page 4-17

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
```

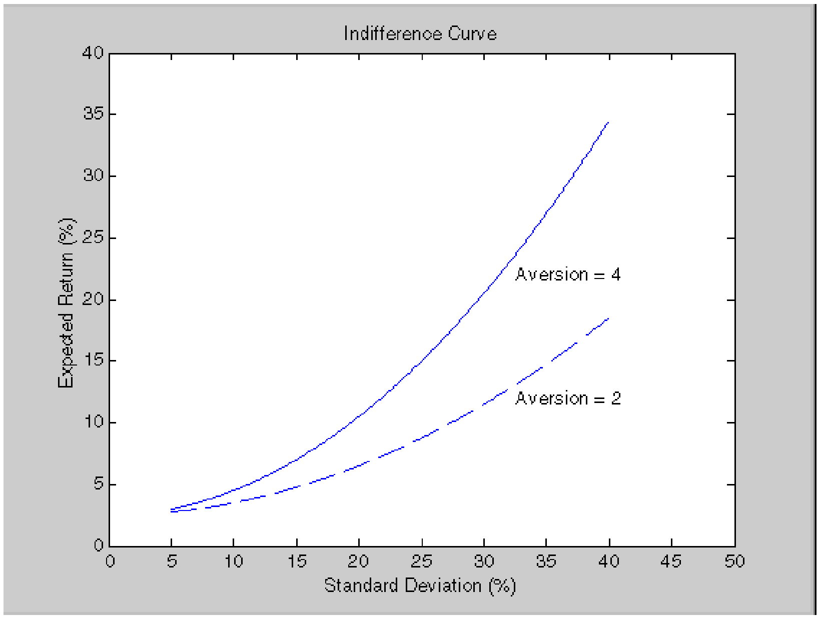
where:

`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 4-17.

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);
```

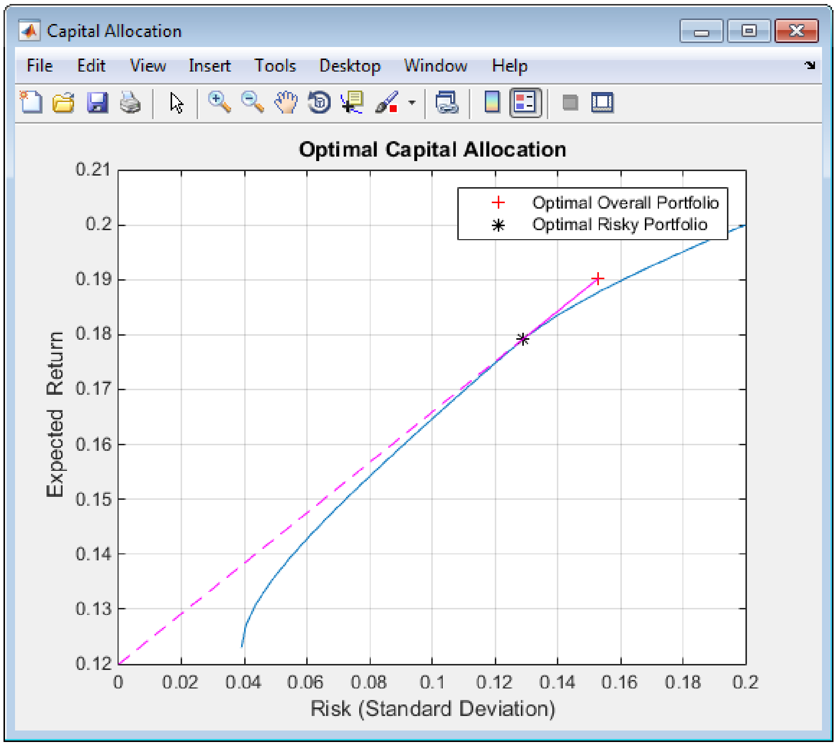
Calling `portopt`, while specifying output arguments, returns the corresponding vectors and arrays representing the risk, return, and weights for each of the portfolios along the efficient frontier. Use these as the first three input arguments to the function `portalloc`.

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

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

Related Examples

- “Portfolio Optimization Functions” on page 3-3
- “Active Returns and Tracking Error Efficient Frontier” on page 3-25
- “Plotting an Efficient Frontier Using portopt” on page 10-22
- “portopt Migration to Portfolio Object” on page 3-11

More About

- “Analyzing Portfolios” on page 3-2
- “Portfolio Object Workflow” on page 4-17

portopt Migration to Portfolio Object

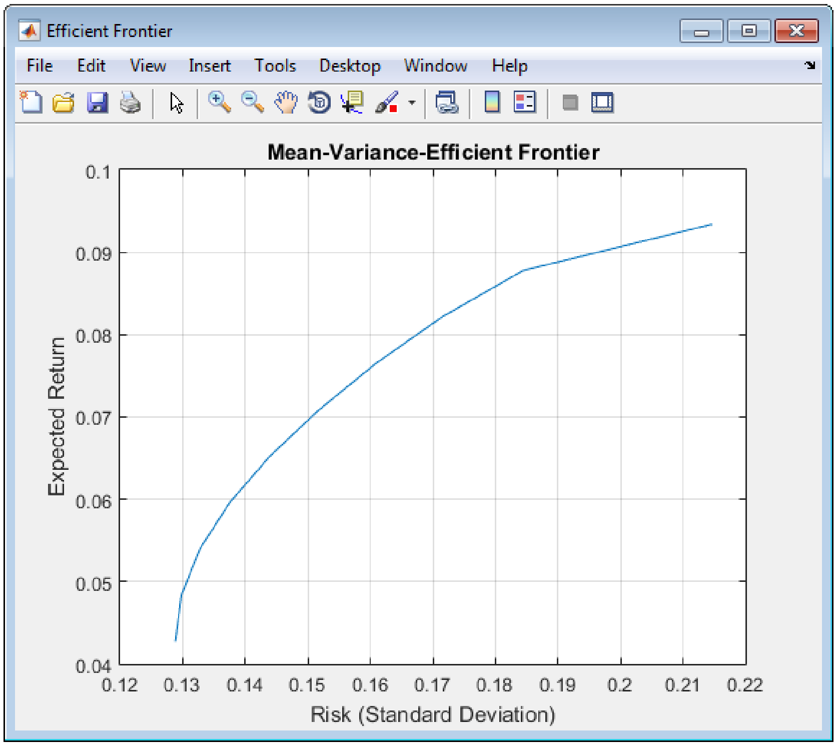
In this section...

Migrate portopt Without Output Arguments

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

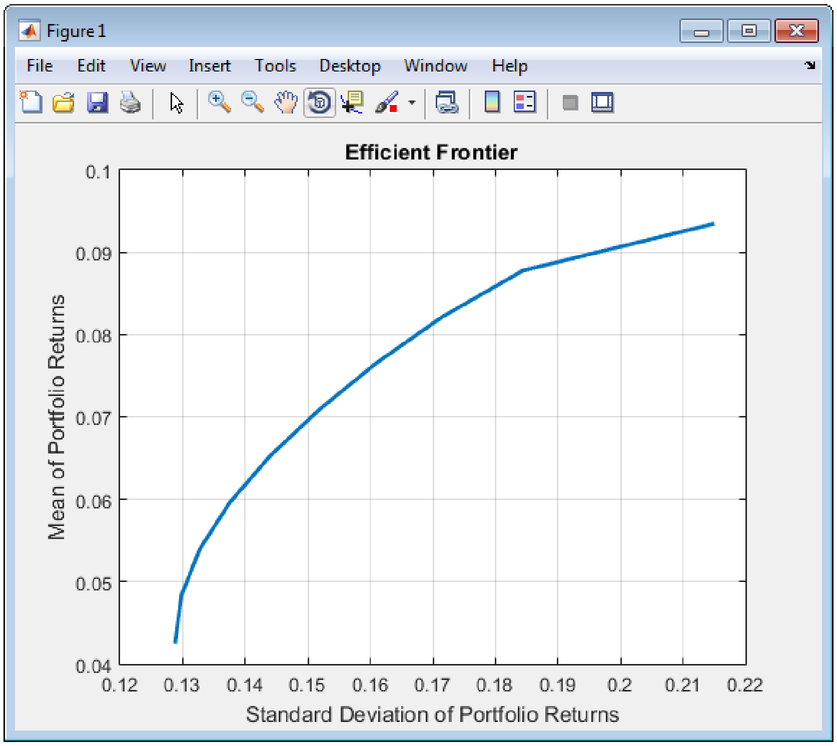
The basic `portopt` functionality is represented as:

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

More About

- “Portfolio Object Workflow” on page 4-17

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.	$W1 + W2$	≥	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

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 3-23).

Specifying Group Constraints

The example above (“Linear Constraint Equations” on page 3-21) 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` object 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 3-21).

See Also

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

Related Examples

- “Setting Default Constraints for Portfolio Weights Using Portfolio Object” on page 4-57
- “Working with 'Simple' Bound Constraints Using Portfolio Object” on page 4-61
- “Working with Budget Constraints Using Portfolio Object” on page 4-64
- “Working with Group Constraints Using Portfolio Object” on page 4-66
- “Working with Group Ratio Constraints Using Portfolio Object” on page 4-69
- “Working with Linear Equality Constraints Using Portfolio Object” on page 4-72
- “Working with Linear Inequality Constraints Using Portfolio Object” on page 4-75
- “Working with Average Turnover Constraints Using Portfolio Object” on page 4-81
- “Working with One-Way Turnover Constraints Using Portfolio Object” on page 4-84
- “Working with Tracking Error Constraints Using Portfolio Object” on page 4-87
- “Asset Allocation Case Study” on page 4-172
- “Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152

More About

- “Portfolio Set for Optimization Using Portfolio Objects” on page 4-8
- “Portfolio Object Workflow” on page 4-17
- “Setting Up a Tracking Portfolio” on page 4-39

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)]
```

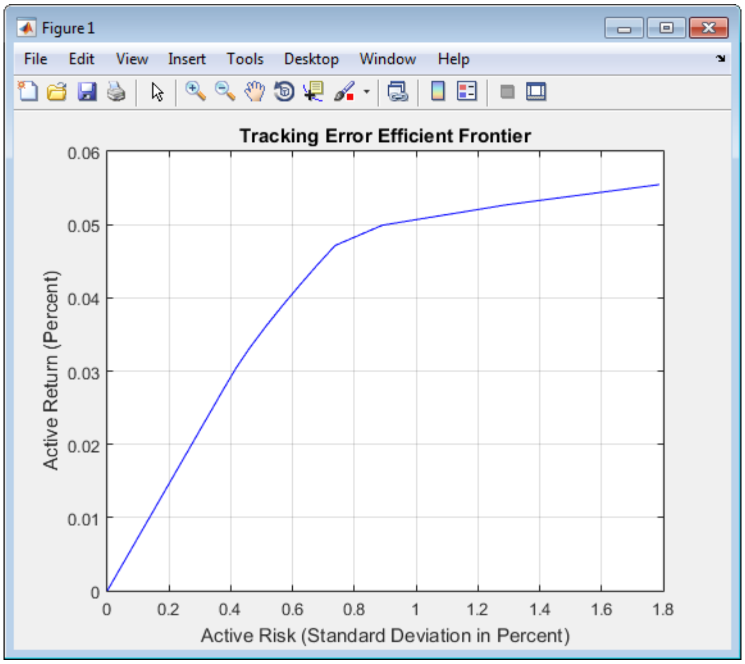
```
ans =

    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

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

Related Examples

- “Portfolio Optimization Functions” on page 3-3
- “Portfolio Selection and Risk Aversion” on page 3-7
- “Plotting an Efficient Frontier Using portopt” on page 10-22

More About

- “Analyzing Portfolios” on page 3-2
- “Portfolio Object Workflow” on page 4-17

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-5) 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 supports mean-variance portfolio optimization (see Markowitz [46], [47] at “Portfolio Optimization” on page A-5). 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 implements what is known as conditional value-at-risk portfolio optimization (see Rockafellar and Uryasev [48], [49] at “Portfolio Optimization” on page A-5), 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 implements what is known as mean-absolute deviation portfolio optimization (see Konno and Yamazaki [50] at “Portfolio Optimization” on page A-5), 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 $\mu : X \rightarrow R$ 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 $y_1,...,y_S$ 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

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

where:

r_0 is the risk-free rate (scalar).

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

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

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

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

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

Risk Proxy

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

Mean Variance

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

$$Variance(x) = x^T C x$$

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-5).

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 –*α*). 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 $s = 1,...,S$. 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

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

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-5.

Mean Absolute-Deviation

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

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

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-5.

See Also

`Portfolio`

Related Examples

- “Creating the Portfolio Object” on page 4-24
- “Working with Portfolio Constraints Using Defaults” on page 4-57
- “Asset Allocation Case Study” on page 4-172
- “Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152

More About

- `Portfolio`
- “Portfolio Object Workflow” on page 4-17
- “Portfolio Set for Optimization Using Portfolio Objects” on page 4-8
- “Default Portfolio Problem” on page 4-16
- “Role of Convexity in Portfolio Problems” on page 4-148

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Portfolio Set for Optimization Using Portfolio Objects

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 (`Portfolio`, `PortfolioCVaR`, and `PortfolioMAD` objects) can have any of these constraints:

- Linear inequality constraints
- Linear equality constraints
- `'Simple'` Bound constraints
- `'Conditional'` Bond constraints
- Budget constraints
- Group constraints
- Group ratio constraints
- Average turnover constraints
- One-way turnover constraints
- Tracking error constraints (only for `Portfolio` object)
- Cardinality constraints

Linear Inequality Constraints

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

$$A_I x \leq b_I$$

where:

x is the portfolio (n vector).

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

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

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

`Portfolio`, `PortfolioCVaR`, and `PortfolioMAD` 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. Use `setEquality` to set linear equality constraints. 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`, `PortfolioCVaR`, and `PortfolioMAD` 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.

'Simple' Bound Constraints

`'Simple'` Bound constraints are specialized linear constraints that confine portfolio weights to fall either above or below specific bounds. Use `setBounds` to specify bound constraints with a `'Simple'` `BoundType`. 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 `'Simple'` 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`, `PortfolioCVaR`, and `PortfolioMAD` 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 4-16) has l_B = `0` with u_B set implicitly through a budget constraint.

'Conditional' Bound Constraints

`'Conditional'` Bound constraints, also called semicontinuous constraints, are mixed-integer linear constraints that confine portfolio weights to fall either above or below specific bounds *if* the asset is selected; otherwise, the value of the asset is zero. Use `setBounds` to specify bound constraints with a `'Conditional' BoundType`. To mathematically formulate this type of constraints, a binary variable v_i is needed. $v_i = 0$ indicates that asset i is not selected and v_i indicates that the asset was selected. Thus

$$l_i v_i \le x_i \le u_i v_i$$

where

x is the portfolio (n vector).

l is the conditional lower-bound constraint (n vector).

u is the conditional upper-bound constraint (n vector).

n is the number of assets in the universe.

`Portfolio`, `PortfolioCVaR`, and `PortfolioMAD` object properties to specify the bound constraint are:

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

The default is to ignore this constraint.

Budget Constraints

Budget constraints are specialized linear constraints that confine the sum of portfolio weights to fall either above or below specific bounds. Use `setBudget` to set budget constraints. 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`, `PortfolioCVaR`, and `PortfolioMAD` 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 4-16) 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 4-51).

Group Constraints

Group constraints are specialized linear constraints that enforce “membership” among groups of assets. Use `setGroups` to set group constraints. The constraints take the form

$$l_G \leq Gx \leq u_G$$

where:

x is the portfolio (n vector).

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

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

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

Each row of G identifies which assets belong to a group associated with that row. Each row contains either `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`, `PortfolioCVaR`, and `PortfolioMAD` 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. Use `setGroupRatio` to set group ratio constraints. The constraints take the form

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

for $i = 1,\ldots, n_R$ 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`, `PortfolioCVaR`, and `PortfolioMAD` 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 compute portfolio turnover as the average of purchases and sales. Use `setTurnover` to set average turnover constraints. Average turnover constraints take the form

$$\frac{1}{2}1^T\left|x - x_0\right| \le \tau$$

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`, `PortfolioCVaR`, and `PortfolioMAD` 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. Use `setOneWayTurnover` to set one-way turnover constraints. 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`, `PortfolioCVaR`, or `PortfolioMAD` 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 4-81) 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. Use `setTrackingError` to set tracking error constraints. The tracking-error constraint has the form

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

where:

x is the portfolio (n vector).

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

C is the covariance of asset returns.

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.

Cardinality Constraints

Cardinality constraint limits the number of assets in the optimal allocation for an `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object . Use `setMinMaxNumAssets` to specify the `'MinNumAssets'` and `'MaxNumAssets'` constraints. To mathematically formulate this type of constraints, a binary variable v_i is needed. $v_i = 0$ indicates that asset i is not selected and $v_i = 1$ indicates that the asset was selected. Thus

$$MinNumAssets \leq \sum_{i=1}^{NumAssets} v_i \leq MaxNumAssets$$

The default is to ignore this constraint.

See Also

`Portfolio` | `PortfolioCVaR` | `PortfolioMAD`

Related Examples

- “Creating the Portfolio Object” on page 4-24
- “Working with Portfolio Constraints Using Defaults” on page 4-57
- “Asset Allocation Case Study” on page 4-172
- “Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-183
- “Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215
- “Portfolio Optimization Using Factor Models” on page 4-224

More About

- `Portfolio`
- “Portfolio Object Workflow” on page 4-17
- “Default Portfolio Problem” on page 4-16

External Websites

- Getting Started with Portfolio Optimization (4 min 13 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` | `PortfolioCVaR` | `PortfolioMAD`

Related Examples

- “Creating the Portfolio Object” on page 4-24
- “Working with Portfolio Constraints Using Defaults” on page 4-57
- “Asset Allocation Case Study” on page 4-172
- “Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-183
- “Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215
- “Portfolio Optimization Using Factor Models” on page 4-224
- “Portfolio Optimization Using Social Performance Measure” on page 4-257
- “Diversify Portfolios Using Custom Objective” on page 4-329

More About

- `Portfolio`
- “Portfolio Set for Optimization Using Portfolio Objects” on page 4-8
- “Portfolio Object Workflow” on page 4-17

External Websites

- Getting Started with Portfolio Optimization (4 min 13 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 4-24.
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 4-41.
3. Specify the Portfolio Constraints.

   Define the constraints for portfolio assets such as linear equality and inequality, bound, budget, group, group ratio, turnover, tracking error, `'Conditional'` `BoundType`, and `MinNumAssets`, `MaxNumAssets` constraints. For more information, see “Working with Portfolio Constraints Using Defaults” on page 4-57 and “Working with 'Conditional' BoundType, MinNumAssets, and MaxNumAssets Constraints Using Portfolio Objects” on page 4-78.
4. Validate the Portfolio.

   Identify errors for the portfolio specification. For more information, see “Validate the Portfolio Problem for Portfolio Object” on page 4-90.
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 4-94 and “Estimate Efficient Frontiers for Portfolio Object” on page 4-118.

   Alternatively, you can estimate an optimal portfolio with a user-defined objective function for a `Portfolio` object. For details on using `estimateCustomObjectivePortfolio` to specify a user-defined objective function, see “Solver Guidelines for Custom Objective Problems Using Portfolio Objects” on page 4-115.
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 4-126.

For an example of this workflow, see “Asset Allocation Case Study” on page 4-172 and “Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152.

See Also

Related Examples

- “Asset Allocation Case Study” on page 4-172
- “Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-183
- “Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215

- “Portfolio Optimization Using Factor Models” on page 4-224
- “Portfolio Optimization Using Social Performance Measure” on page 4-257
- “Diversify Portfolios Using Custom Objective” on page 4-329

More About

- “Portfolio Optimization Theory” on page 4-3

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 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 `Portfolio` 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 `Portfolio` to set the properties for the `Portfolio` object. `Portfolio` 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 4-24.

Setting and Getting Properties

You can set properties of a `Portfolio` object using either `Portfolio` 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 object 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 Portfolio, 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 object and the set functions have several useful features:

- Portfolio and the set functions try to determine the dimensions of your problem with either explicit or implicit inputs.
- Portfolio and the set functions try to resolve ambiguities with default choices.
- Portfolio and the 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. An efficient portfolio are the portfolios that satisfy the criteria of minimum risk for a given level of return and maximum return for a given level of risk. 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 attains targeted values for return proxies
- That attains 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 enable 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)
```

```
disp(p)
  3×2 Portfolio array 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
    MinNumAssets
    MaxNumAssets
    BoundType
```

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 `Portfolio` 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 4-24
- “Working with Portfolio Constraints Using Defaults” on page 4-57
- “Asset Allocation Case Study” on page 4-172
- “Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-183
- “Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215

- “Portfolio Optimization Using Factor Models” on page 4-224
- “Portfolio Optimization Using Social Performance Measure” on page 4-257
- “Diversify Portfolios Using Custom Objective” on page 4-329

More About

- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-17

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)

Creating the Portfolio Object

In this section...

To create a fully specified mean-variance portfolio optimization problem, instantiate the `Portfolio` object using `Portfolio`. For information on the workflow when using `Portfolio` objects, see "Portfolio Object Workflow" on page 4-17.

Syntax

Use `Portfolio` to create an instance of an object of the `Portfolio` class. You can use `Portfolio` 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` object also accepts collections of argument name-value pair arguments for properties and their values. The `Portfolio` object 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 `Portfolio` 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 4-29). The `Portfolio` object detects 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-5.

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 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: []
     MinNumAssets: []
     MaxNumAssets: []
        BoundType: []
```

The approaches listed provide a way to set up a portfolio optimization problem with the `Portfolio` object. 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` object to directly set up a “standard” portfolio optimization problem, given a mean and covariance of asset returns in the variables `m` and `C`:

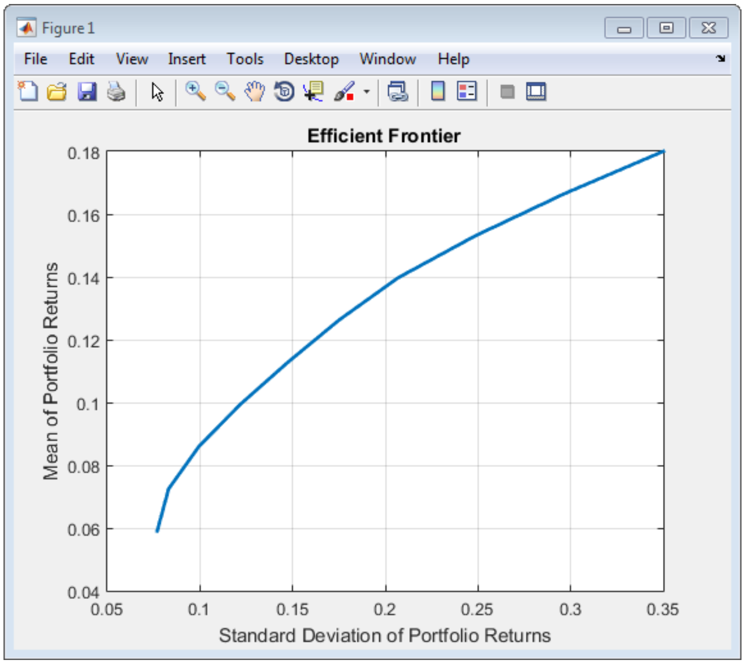
```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];

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

```
p = 

  Portfolio with properties:

          BuyCost: []
         SellCost: []
     RiskFreeRate: []
        AssetMean: [4×1 double]
       AssetCovar: [4×4 double]
    TrackingError: []
     TrackingPort: []
         Turnover: []
      BuyTurnover: []
     SellTurnover: []
             Name: []
        NumAssets: 4
        AssetList: []
         InitPort: []
      AInequality: []
      bInequality: []
        AEquality: []
        bEquality: []
       LowerBound: [4×1 double]
       UpperBound: []
      LowerBudget: 1
      UpperBudget: 1
      GroupMatrix: []
       LowerGroup: []
       UpperGroup: []
           GroupA: []
           GroupB: []
       LowerRatio: []
       UpperRatio: []
     MinNumAssets: []
     MaxNumAssets: []
        BoundType: []
```

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`.

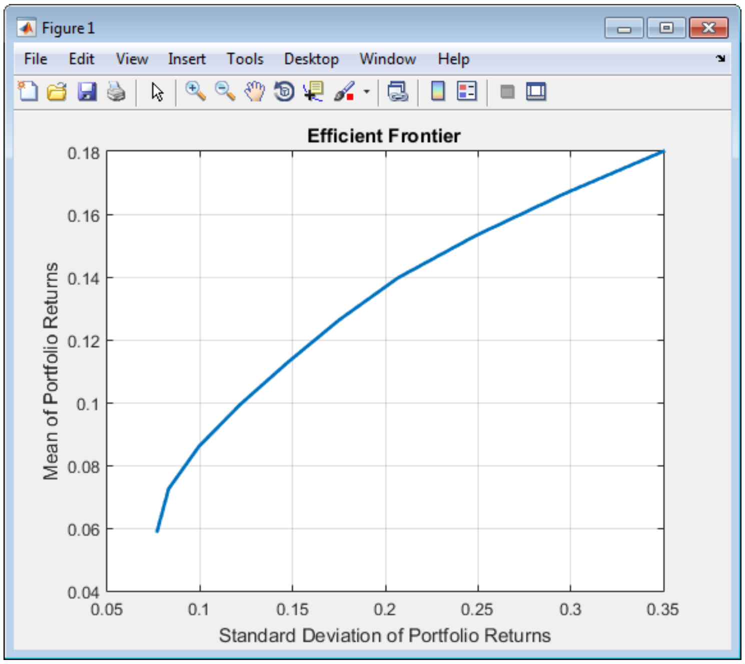
```
p.plotFrontier
```



Using the Portfolio Function with a Sequence of Steps

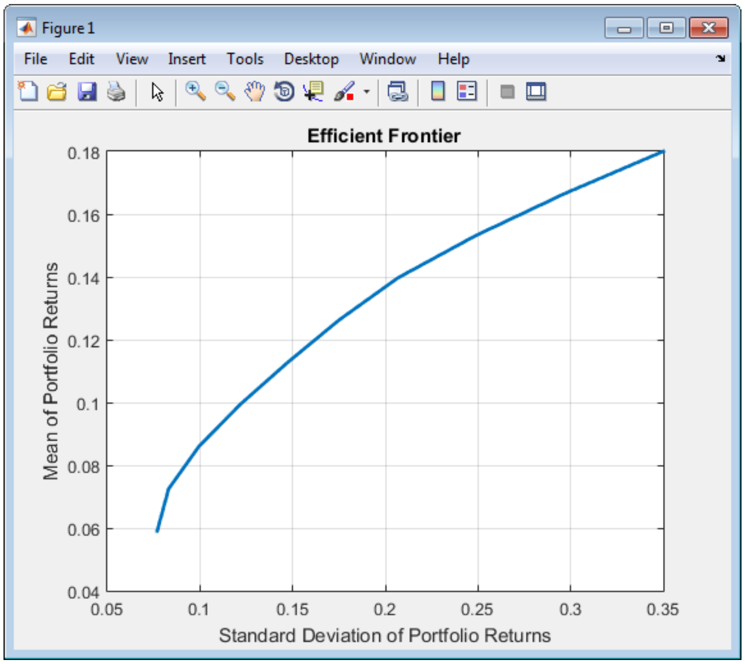
An alternative way to accomplish the same task of setting up a “standard” portfolio optimization problem, given a mean and covariance of asset returns in the variables `m` and `C` (which also illustrates that argument names are not case-sensitive):

```
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 `Portfolio` 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` object 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` object 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` object 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` object. 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 `Portfolio` 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 4-32
- “Working with Portfolio Constraints Using Defaults” on page 4-57
- “Asset Allocation Case Study” on page 4-172
- “Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-183
- “Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215
- “Portfolio Optimization Using Factor Models” on page 4-224
- “Diversify Portfolios Using Custom Objective” on page 4-329
- “Portfolio Optimization Using Social Performance Measure” on page 4-257

More About

- “Portfolio Object” on page 4-19
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-17

External Websites

- Getting Started with Portfolio Optimization (4 min 13 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 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` object:

```
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 object 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 object, 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 object:

```
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 | setAssetList | setInitPort | setTrackingPort | estimateBounds | checkFeasibility

Related Examples

- “Setting Up an Initial or Current Portfolio” on page 4-36
- “Working with Portfolio Constraints Using Defaults” on page 4-57
- “Asset Returns and Moments of Asset Returns Using Portfolio Object” on page 4-41
- “Validate the Portfolio Problem for Portfolio Object” on page 4-90
- “Asset Allocation Case Study” on page 4-172
- “Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-183

- “Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215
- “Portfolio Optimization Using Factor Models” on page 4-224
- “Portfolio Optimization Using Social Performance Measure” on page 4-257
- “Diversify Portfolios Using Custom Objective” on page 4-329

More About

- “Portfolio Object” on page 4-19
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-17

External Websites

- Getting Started with Portfolio Optimization (4 min 13 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 4-91. Suppose that you have an initial portfolio in x0, then use the Portfolio object 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 object 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 = 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 4-81, "Working with One-Way Turnover Constraints Using Portfolio Object" on page 4-84, and "Working with Transaction Costs" on page 4-53 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` object 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` | `setAssetList` | `setInitPort` | `estimateBounds` | `checkFeasibility`

Related Examples

- "Setting Up a Tracking Portfolio" on page 4-39
- "Common Operations on the Portfolio Object" on page 4-32
- "Working with Portfolio Constraints Using Defaults" on page 4-57
- "Asset Returns and Moments of Asset Returns Using Portfolio Object" on page 4-41
- "Validate the Portfolio Problem for Portfolio Object" on page 4-90
- "Asset Allocation Case Study" on page 4-172
- "Portfolio Optimization Examples Using Financial Toolbox™" on page 4-152
- "Portfolio Optimization with Semicontinuous and Cardinality Constraints" on page 4-183
- "Black-Litterman Portfolio Optimization Using Financial Toolbox™" on page 4-215

- “Portfolio Optimization Using Factor Models” on page 4-224
- “Portfolio Optimization Using Social Performance Measure” on page 4-257
- “Diversify Portfolios Using Custom Objective” on page 4-329

More About

- “Portfolio Object” on page 4-19
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-17

External Websites

- Getting Started with Portfolio Optimization (4 min 13 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 4-87.

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` object 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` object 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` | `setAssetList` | `setInitPort` | `setTrackingPort` | `setTrackingError` | `estimateBounds` | `checkFeasibility`

Related Examples

- “Setting Up an Initial or Current Portfolio” on page 4-36
- “Common Operations on the Portfolio Object” on page 4-32
- “Working with Portfolio Constraints Using Defaults” on page 4-57
- “Asset Returns and Moments of Asset Returns Using Portfolio Object” on page 4-41
- “Validate the Portfolio Problem for Portfolio Object” on page 4-90
- “Asset Allocation Case Study” on page 4-172
- “Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-183
- “Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215
- “Portfolio Optimization Using Factor Models” on page 4-224
- “Diversify Portfolios Using Custom Objective” on page 4-329

More About

- “Portfolio Object” on page 4-19
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-17

External Websites

- Getting Started with Portfolio Optimization (4 min 13 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 4-17.

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` 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;
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` object 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` object and the `setAssetMoments` function perform scalar expansion on arguments for the moments of asset returns. When using the `Portfolio` object, 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` object 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 `table` or `timetable` (see “Estimating Asset Moments from Time Series Data” on page 4-47). 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` object, 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 a table or timetable. estimateAssetMoments implicitly works with matrices of data or data in a table or timetable object using the same rules for whether the data are returns or prices.

To illustrate the use of a table and timetable, use array2table and array2timetable to create a table and a timetable that contain asset returns generated with portsim (see “Estimating Asset

Moments from Prices or Returns” on page 4-44). Two portfolio objects are then created with the `AssetReturns` based on a table and a timetable 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;

assetRetnScenarios = portsim(m', C, 120);
dates = datetime(datenum(2001,1:120,31), 'ConvertFrom', 'datenum');
assetsName = {'Bonds', 'LargeCap', 'SmallCap', 'Emerging'};
assetRetnTimeTable = array2timetable(assetRetnScenarios,'RowTimes',dates, 'VariableNames', assetsName);
assetRetnTable = array2table(assetRetnScenarios, 'VariableNames', assetsName);


% Create two Portfolio objects:
% p with predefined mean and covar: q with asset return scenarios to estimate mean and covar.
p = Portfolio('mean', m, 'covar', C);
q = Portfolio;

% estimate asset moments with timetable
q = estimateAssetMoments(q, assetRetnTimeTable);
[passetmean, passetcovar] = getAssetMoments(p)
[qassetmean, qassetcovar] = getAssetMoments(q)

% estimate asset moments with table
q = estimateAssetMoments(q, assetRetnTable);
[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


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 between th two portfolios. In addition, `estimateAssetMoments` also extracts asset names or identifiers from a `table` or `timetable` when 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. To show this, the formation of the `Portfolio` object `q` is repeated from the previous example with the `'GetAssetList'` flag set to `true` extracts the column labels from a table or timetable object:

```
q = estimateAssetMoments(q,assetRetnTable,'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 4-32.

See Also

`Portfolio` | `setAssetMoments` | `estimateAssetMoments` | `getAssetMoments` | `setCosts`

Related Examples

- “Creating the Portfolio Object” on page 4-24
- “Working with Portfolio Constraints Using Defaults” on page 4-57
- “Validate the Portfolio Problem for Portfolio Object” on page 4-90
- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-94

- “Estimate Efficient Frontiers for Portfolio Object” on page 4-118
- “Asset Allocation Case Study” on page 4-172
- “Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-183
- “Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215
- “Portfolio Optimization Using Factor Models” on page 4-224
- “Portfolio Optimization Using Social Performance Measure” on page 4-257
- “Diversify Portfolios Using Custom Objective” on page 4-329

More About

- “Portfolio Object” on page 4-19
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-17

External Websites

- Getting Started with Portfolio Optimization (4 min 13 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 object:

```
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 | setAssetMoments | estimateAssetMoments | getAssetMoments

Related Examples

- “Creating the Portfolio Object” on page 4-24
- “Working with Portfolio Constraints Using Defaults” on page 4-57
- “Validate the Portfolio Problem for Portfolio Object” on page 4-90
- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-94
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-118
- “Asset Allocation Case Study” on page 4-172
- “Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-183
- “Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215
- “Portfolio Optimization Using Factor Models” on page 4-224
- “Portfolio Optimization Using Social Performance Measure” on page 4-257
- “Diversify Portfolios Using Custom Objective” on page 4-329

More About

- “Portfolio Object” on page 4-19
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-17

External Websites

- Getting Started with Portfolio Optimization (4 min 13 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 4-4). 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` object. 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” on page 4-204.

Setting Transaction Costs with Scalar Expansion

Both the `Portfolio` object 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` object 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` | `setAssetMoments` | `estimateAssetMoments` | `getAssetMoments` | `setCosts`

Related Examples

- “Creating the Portfolio Object” on page 4-24
- “Working with Portfolio Constraints Using Defaults” on page 4-57
- “Validate the Portfolio Problem for Portfolio Object” on page 4-90
- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-94
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-118
- “Portfolio Analysis with Turnover Constraints” on page 4-204

- “Asset Allocation Case Study” on page 4-172
- “Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-183
- “Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215
- “Portfolio Optimization Using Factor Models” on page 4-224
- “Portfolio Optimization Using Social Performance Measure” on page 4-257
- “Diversify Portfolios Using Custom Objective” on page 4-329

More About

- “Portfolio Object” on page 4-19
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-17

External Websites

- Getting Started with Portfolio Optimization (4 min 13 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 4-17.

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` object 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: [20×1 double]
       UpperBound: []
      LowerBudget: 1
      UpperBudget: 1
      GroupMatrix: []
       LowerGroup: []
       UpperGroup: []
           GroupA: []
           GroupB: []
       LowerRatio: []
       UpperRatio: []
        BoundType: []
     MinNumAssets: []
     MaxNumAssets: []
```

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: [20×1 double]
       UpperBound: []
      LowerBudget: 1
      UpperBudget: 1
      GroupMatrix: []
       LowerGroup: []
       UpperGroup: []
           GroupA: []
           GroupB: []
       LowerRatio: []
       UpperRatio: []
        BoundType: [0×0 categorical]
     MinNumAssets: []
     MaxNumAssets: []
```

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: [20×1 double]
       UpperBound: []
      LowerBudget: 1
      UpperBudget: 1
      GroupMatrix: []
       LowerGroup: []
       UpperGroup: []
           GroupA: []
           GroupB: []
       LowerRatio: []
       UpperRatio: []
        BoundType: [0×0 categorical]
     MinNumAssets: []
     MaxNumAssets: []
```

See Also

`Portfolio` | `setDefaultConstraints` | `setBounds` | `setBudget` | `setGroups` | `setGroupRatio` | `setEquality` | `setInequality` | `setTurnover` | `setOneWayTurnover` | `setTrackingPort` | `setTrackingError`

Related Examples

- “Working with 'Simple' Bound Constraints Using Portfolio Object” on page 4-61
- “Working with Budget Constraints Using Portfolio Object” on page 4-64
- “Working with Group Constraints Using Portfolio Object” on page 4-66
- “Working with Group Ratio Constraints Using Portfolio Object” on page 4-69
- “Working with Linear Equality Constraints Using Portfolio Object” on page 4-72

- “Working with Linear Inequality Constraints Using Portfolio Object” on page 4-75
- “Working with Average Turnover Constraints Using Portfolio Object” on page 4-81
- “Working with One-Way Turnover Constraints Using Portfolio Object” on page 4-84
- “Working with Tracking Error Constraints Using Portfolio Object” on page 4-87
- “Working with 'Conditional' BoundType, MinNumAssets, and MaxNumAssets Constraints Using Portfolio Objects” on page 4-78
- “Creating the Portfolio Object” on page 4-24
- “Validate the Portfolio Problem for Portfolio Object” on page 4-90
- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-94
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-118
- “Constraint Specification Using a Portfolio Object” on page 3-19
- “Asset Allocation Case Study” on page 4-172
- “Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-183
- “Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215
- “Portfolio Optimization Using Factor Models” on page 4-224
- “Portfolio Optimization Using Social Performance Measure” on page 4-257
- “Diversify Portfolios Using Custom Objective” on page 4-329

More About

- “Portfolio Object” on page 4-19
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-17
- “Setting Up a Tracking Portfolio” on page 4-39

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Working with 'Simple' Bound Constraints Using Portfolio Object

'Simple' bound constraints are optional linear constraints that maintain upper and lower bounds on portfolio weights (see “'Simple' Bound Constraints” on page 4-9). Although every portfolio set must be bounded, it is not necessary to specify a portfolio set with explicit bound constraints. For example, you can create a portfolio set with an implicit upper bound constraint or a portfolio set with 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 4-57).

Setting 'Simple' Bounds Using the Portfolio Function

The properties for bound constraints are set through the Portfolio object. 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, 'BoundType', 'Simple');
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 4-64.

Setting 'Simple' 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,'BoundType', 'Simple');
disp(p.NumAssets)
disp(p.LowerBound)
disp(p.UpperBound)
```

```
  2

  0.5000
  0.2500
```

```
    0.7500
    0.5000
```

Setting 'Simple' Bounds Using the Portfolio Function or setBounds Function

Both the `Portfolio` object 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,'BoundType', 'Simple');
```

or

```
p = Portfolio('NumAssets', 500);
p = setBounds(p, 0, 0.05,'BoundType','Simple');
```

or

```
p = Portfolio;
p = setBounds(p, 0, 0.05, 'NumAssets', 500,'BoundType','Simple');
```

To clear bound constraints from your `Portfolio` object, use either the `Portfolio` object 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` | `setDefaultConstraints` | `setBounds` | `setBudget` | `setGroups` | `setGroupRatio` | `setEquality` | `setInequality` | `setTurnover` | `setOneWayTurnover` | `setTrackingPort` | `setTrackingError`

Related Examples

- “Creating the Portfolio Object” on page 4-24
- “Working with 'Conditional' BoundType, MinNumAssets, and MaxNumAssets Constraints Using Portfolio Objects” on page 4-78
- “Working with Portfolio Constraints Using Defaults” on page 4-57
- “Validate the Portfolio Problem for Portfolio Object” on page 4-90
- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-94
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-118
- “Constraint Specification Using a Portfolio Object” on page 3-19
- “Asset Allocation Case Study” on page 4-172
- “Portfolio Optimization Examples Using Financial Toolbox™ ” on page 4-152
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-183

- “Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215
- “Portfolio Optimization Using Factor Models” on page 4-224
- “Portfolio Optimization Using Social Performance Measure” on page 4-257
- “Diversify Portfolios Using Custom Objective” on page 4-329

More About

- “Portfolio Object” on page 4-19
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-17
- “Setting Up a Tracking Portfolio” on page 4-39

External Websites

- Getting Started with Portfolio Optimization (4 min 13 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 4-10). Budget constraints have properties `LowerBudget` for the lower budget constraint and `UpperBudget` for the upper budget constraint. If you set up a portfolio optimization problem that requires portfolios to be fully invested in your universe of assets, you can set `LowerBudget` to be equal to `UpperBudget`. These budget constraints can be set with default values equal to `1` using `setDefaultConstraints` (see “Setting Default Constraints for Portfolio Weights Using Portfolio Object” on page 4-57).

Setting Budget Constraints Using the Portfolio Function

The properties for the budget constraint can also be set using the `Portfolio` object. 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 4-51. To clear either bound for the budget constraint from your Portfolio object, use either the `Portfolio` object 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` | `setDefaultConstraints` | `setBounds` | `setBudget` | `setGroups` | `setGroupRatio` | `setEquality` | `setInequality` | `setTurnover` | `setOneWayTurnover` | `setTrackingPort` | `setTrackingError`

Related Examples

- “Creating the Portfolio Object” on page 4-24
- “Working with Portfolio Constraints Using Defaults” on page 4-57
- “Validate the Portfolio Problem for Portfolio Object” on page 4-90
- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-94
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-118
- “Constraint Specification Using a Portfolio Object” on page 3-19
- “Asset Allocation Case Study” on page 4-172
- “Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-183
- “Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215
- “Portfolio Optimization Using Factor Models” on page 4-224
- “Portfolio Optimization Using Social Performance Measure” on page 4-257
- “Diversify Portfolios Using Custom Objective” on page 4-329

More About

- “Portfolio Object” on page 4-19
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-17
- “Setting Up a Tracking Portfolio” on page 4-39

External Websites

- Getting Started with Portfolio Optimization (4 min 13 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 4-11). Although the constraints are implemented as general constraints, the usual convention is to form a group matrix that identifies membership of each asset within a specific group with Boolean indicators (either `true` or `false` or with `1` or `0`) for each element in the group matrix. Group constraints have properties `GroupMatrix` for the group membership matrix, `LowerGroup` for the lower-bound constraint on groups, and `UpperGroup` for the upper-bound constraint on groups.

Setting Group Constraints Using the Portfolio Function

The properties for group constraints are set through the `Portfolio` object. 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
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` object 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` | `setDefaultConstraints` | `setBounds` | `setBudget` | `setGroups` | `setGroupRatio` | `setEquality` | `setInequality` | `setTurnover` | `setOneWayTurnover` | `setTrackingPort` | `setTrackingError`

Related Examples

- "Creating the Portfolio Object" on page 4-24
- "Working with Portfolio Constraints Using Defaults" on page 4-57
- "Validate the Portfolio Problem for Portfolio Object" on page 4-90
- "Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object" on page 4-94
- "Estimate Efficient Frontiers for Portfolio Object" on page 4-118
- "Constraint Specification Using a Portfolio Object" on page 3-19
- "Asset Allocation Case Study" on page 4-172
- "Portfolio Optimization Examples Using Financial Toolbox™" on page 4-152
- "Portfolio Optimization with Semicontinuous and Cardinality Constraints" on page 4-183
- "Black-Litterman Portfolio Optimization Using Financial Toolbox™" on page 4-215
- "Portfolio Optimization Using Factor Models" on page 4-224
- "Portfolio Optimization Using Social Performance Measure" on page 4-257
- "Diversify Portfolios Using Custom Objective" on page 4-329

More About

- "Portfolio Object" on page 4-19
- "Portfolio Optimization Theory" on page 4-3
- "Portfolio Object Workflow" on page 4-17
- "Setting Up a Tracking Portfolio" on page 4-39

External Websites

- Getting Started with Portfolio Optimization (4 min 13 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 4-12). Although the constraints are implemented as general constraints, the usual convention is to specify a pair of group matrices that identify membership of each asset within specific groups with Boolean indicators (either `true` or `false` or with `1` or `0`) for each element in each of the group matrices. The goal is to ensure that the ratio of a base group 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` object. 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 nonfinancial companies (assets 4–6). To set group ratio constraints:

```
GA = [ 1 1 1 0 0 0 ];    % financial companies
GB = [ 0 0 0 1 1 1 ];    % nonfinancial companies
p = Portfolio('GroupA', GA, 'GroupB', GB, 'UpperRatio', 0.5);
disp(p.NumAssets)
disp(p.GroupA)
disp(p.GroupB)
disp(p.UpperRatio)
```

```
6

1     1     1     0     0     0

0     0     0     1     1     1

0.5000
```

Group matrices GA and GB in this example can be logical matrices with `true` and `false` elements that yield the same result:

```
GA = [ true true true false false false ];    % financial companies
GB = [ false false 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 nonfinancial companies (assets 4–6). Given a `Portfolio` object `p`, use `setGroupRatio` to set the group constraints:

```
GA = [ true true true false false false ];   % financial companies
GB = [ false false false true true true ];   % nonfinancial companies
p = Portfolio;
p = setGroupRatio(p, GA, GB, [], 0.5);
disp(p.NumAssets)
disp(p.GroupA)
disp(p.GroupB)
disp(p.UpperRatio)
```

```
6

1     1     1     0     0     0

0     0     0     1     1     1

0.5000
```

In this example, you would set the `LowerRatio` property to be empty (`[]`).

Suppose 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 object, 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 | setDefaultConstraints | setBounds | setBudget | setGroups | setGroupRatio | setEquality | setInequality | setTurnover | setOneWayTurnover | setTrackingPort | setTrackingError

Related Examples

- “Creating the Portfolio Object” on page 4-24
- “Working with Portfolio Constraints Using Defaults” on page 4-57
- “Validate the Portfolio Problem for Portfolio Object” on page 4-90
- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-94
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-118
- “Constraint Specification Using a Portfolio Object” on page 3-19
- “Asset Allocation Case Study” on page 4-172
- “Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-183
- “Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215
- “Portfolio Optimization Using Factor Models” on page 4-224
- “Portfolio Optimization Using Social Performance Measure” on page 4-257
- “Diversify Portfolios Using Custom Objective” on page 4-329

More About

- “Portfolio Object” on page 4-19
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-17
- “Setting Up a Tracking Portfolio” on page 4-39

External Websites

- Getting Started with Portfolio Optimization (4 min 13 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 4-9). Linear equality constraints have properties AEquality, for the equality constraint matrix, and bEquality, for the equality constraint vector.

Setting Linear Equality Constraints Using the Portfolio Function

The properties for linear equality constraints are set using the Portfolio object. 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` object, `setEquality`, and `addEquality` implement scalar expansion on the `bEquality` property based on the dimension of the matrix in the `AEquality` property.

See Also

`Portfolio` | `setDefaultConstraints` | `setBounds` | `setBudget` | `setGroups` | `setGroupRatio` | `setEquality` | `setInequality` | `setTurnover` | `setOneWayTurnover` | `setTrackingPort` | `setTrackingError`

Related Examples

- “Creating the Portfolio Object” on page 4-24
- “Working with Portfolio Constraints Using Defaults” on page 4-57
- “Validate the Portfolio Problem for Portfolio Object” on page 4-90
- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-94
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-118
- “Constraint Specification Using a Portfolio Object” on page 3-19
- “Asset Allocation Case Study” on page 4-172
- “Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-183
- “Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215
- “Portfolio Optimization Using Factor Models” on page 4-224
- “Portfolio Optimization Using Social Performance Measure” on page 4-257
- “Diversify Portfolios Using Custom Objective” on page 4-329

More About

- “Portfolio Object” on page 4-19
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-17

- “Setting Up a Tracking Portfolio” on page 4-39

External Websites

- Getting Started with Portfolio Optimization (4 min 13 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 4-8). Linear inequality constraints have properties AInequality for the inequality constraint matrix, and bInequality for the inequality constraint vector.

Setting Linear Inequality Constraints Using the Portfolio Function

The properties for linear inequality constraints are set using the Portfolio object. 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` object, `setInequality`, and `addInequality` implement scalar expansion on the `bInequality` property based on the dimension of the matrix in the `AInequality` property.

See Also

`Portfolio` | `setDefaultConstraints` | `setBounds` | `setBudget` | `setGroups` | `setGroupRatio` | `setEquality` | `setInequality` | `setTurnover` | `setOneWayTurnover` | `setTrackingPort` | `setTrackingError`

Related Examples

- “Creating the Portfolio Object” on page 4-24
- “Working with Portfolio Constraints Using Defaults” on page 4-57
- “Validate the Portfolio Problem for Portfolio Object” on page 4-90
- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-94
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-118
- “Constraint Specification Using a Portfolio Object” on page 3-19
- “Asset Allocation Case Study” on page 4-172
- “Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-183
- “Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215
- “Portfolio Optimization Using Factor Models” on page 4-224
- “Portfolio Optimization Using Social Performance Measure” on page 4-257
- “Diversify Portfolios Using Custom Objective” on page 4-329

More About

- “Portfolio Object” on page 4-19
- “Portfolio Optimization Theory” on page 4-3

- “Portfolio Object Workflow” on page 4-17
- “Setting Up a Tracking Portfolio” on page 4-39

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Working with 'Conditional' BoundType, MinNumAssets, and MaxNumAssets Constraints Using Portfolio Objects

When any one, or any combination of `'Conditional'` `BoundType`, `MinNumAssets`, or `MaxNumAssets` constraints are active, the portfolio problem is formulated by adding `NumAssets` binary variables, where `0` indicates not invested, and `1` is invested. For example, to explain the `'Conditional'` `BoundType` and `MinNumAssets` and `MaxNumAssets` constraints, assume that your portfolio has a universe of 100 assets that you want to invest:

- `'Conditional'` `BoundType` (also known as semicontinuous constraints), set by `setBounds`, is often used in situations where you do not want to invest small values. A standard example is a portfolio optimization problem where many small allocations are not attractive because of transaction costs. Instead, you prefer fewer instruments in the portfolio with larger allocations. This situation can be handled using`'Conditional'` `BoundType` constraints for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object.

  For example, the weight you invest in each asset is either `0` or between `[0.01, 0.5]`. Generally, a semicontinuous variable *x* is a continuous variable between bounds [`lb`, `ub`] that also can assume the value `0`, where `lb` > `0`, `lb` ≤ `ub`. Applying this to portfolio optimization requires that very small or large positions should be avoided, that is values that fall in (`0`, `lb`) or are more than `ub`.
- `MinNumAssets` and `MaxNumAssets` (also known as cardinality constraints), set by `setMinMaxNumAssets`, limit the number of assets in a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For example, if you have 100 assets in your portfolio and you want the number of assets allocated in the portfolio to be from 40 through 60. Using `MinNumAssets` and `MaxNumAssets` you can limit the number of assets in the optimized portfolio, which allows you to limit transaction and operational costs or to create an index tracking portfolio.

Setting 'Conditional' BoundType Constraints Using the setBounds Function

Use `setBounds` with a `'conditional'` `BoundType` to set *xi* = `0` or `0.02` <= *xi* <= `0.5` for all *i*=1,...`NumAssets`:

```
p = Portfolio('AssetMean', AssetMean, 'AssetCovar', AssetCovar);
p = setBounds(p, 0.02, 0.5,'BoundType', 'Conditional', 'NumAssets', 3)
```

```
p = 
  Portfolio with properties:

          BuyCost: []
         SellCost: []
     RiskFreeRate: []
        AssetMean: [3×1 double]
       AssetCovar: [3×3 double]
    TrackingError: []
     TrackingPort: []
         Turnover: []
      BuyTurnover: []
     SellTurnover: []
             Name: []
        NumAssets: 3
        AssetList: []
```

```
         InitPort: []
      AInequality: []
      bInequality: []
        AEquality: []
        bEquality: []
       LowerBound: [3×1 double]
       UpperBound: [3×1 double]
      LowerBudget: []
      UpperBudget: []
      GroupMatrix: []
       LowerGroup: []
       UpperGroup: []
           GroupA: []
           GroupB: []
       LowerRatio: []
       UpperRatio: []
        BoundType: [3×1 categorical]
     MinNumAssets: []
     MaxNumAssets: []
```

Setting the Limits on the Number of Assets Invested Using the setMinMaxNumAssets Function

You can also set the `MinNumAssets` and `MaxNumAssets` properties to define a limit on the number of assets invested using `setMinMaxNumAssets`. For example, by setting `MinNumAssets=MaxNumAssets=2`, only two of the three assets are invested in the portfolio.

```
AssetMean = [ 0.0101110; 0.0043532; 0.0137058 ];
AssetCovar = [ 0.00324625 0.00022983 0.00420395;
               0.00022983 0.00049937 0.00019247;
               0.00420395 0.00019247 0.00764097 ];

p = Portfolio('AssetMean', AssetMean, 'AssetCovar', AssetCovar);
p = setMinMaxNumAssets(p, 2, 2)
```

```
 Portfolio with properties:

          BuyCost: []
         SellCost: []
     RiskFreeRate: []
        AssetMean: [3×1 double]
       AssetCovar: [3×3 double]
    TrackingError: []
     TrackingPort: []
         Turnover: []
      BuyTurnover: []
     SellTurnover: []
             Name: []
        NumAssets: 3
        AssetList: []
         InitPort: []
      AInequality: []
      bInequality: []
        AEquality: []
        bEquality: []
       LowerBound: []
       UpperBound: []
```

```
         LowerBudget: []
         UpperBudget: []
         GroupMatrix: []
          LowerGroup: []
          UpperGroup: []
              GroupA: []
              GroupB: []
          LowerRatio: []
          UpperRatio: []
           BoundType: []
        MinNumAssets: 2
        MaxNumAssets: 2
```

See Also

`Portfolio` | `setBounds` | `setMinMaxNumAssets` | `setDefaultConstraints` | `setBounds` | `setBudget` | `setGroups` | `setGroupRatio` | `setEquality` | `setInequality` | `setTurnover` | `setOneWayTurnover` | `setTrackingPort` | `setTrackingError`

Related Examples

- “Creating the Portfolio Object” on page 4-24
- “Working with Portfolio Constraints Using Defaults” on page 4-57
- “Troubleshooting for Setting 'Conditional' BoundType, MinNumAssets, and MaxNumAssets Constraints” on page 4-139
- “Validate the Portfolio Problem for Portfolio Object” on page 4-90
- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-94
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-118
- “Constraint Specification Using a Portfolio Object” on page 3-19
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-183
- “Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215
- “Portfolio Optimization Using Factor Models” on page 4-224
- “Portfolio Optimization Using Social Performance Measure” on page 4-257
- “Diversify Portfolios Using Custom Objective” on page 4-329

More About

- “Portfolio Object” on page 4-19
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-17
- “Setting Up a Tracking Portfolio” on page 4-39

External Websites

- Getting Started with Portfolio Optimization (4 min 13 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 4-12) that enforces an upper bound on the average of purchases and sales. The turnover constraint can be set using the `Portfolio` object 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` object. 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 4-36).

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” on page 4-204.

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 object or setTurnover with empty inputs for the properties to be cleared.

See Also

Portfolio | setDefaultConstraints | setBounds | setBudget | setGroups | setGroupRatio | setEquality | setInequality | setTurnover | setOneWayTurnover | setTrackingPort | setTrackingError

Related Examples

- “Creating the Portfolio Object” on page 4-24
- “Working with Portfolio Constraints Using Defaults” on page 4-57

- “Validate the Portfolio Problem for Portfolio Object” on page 4-90
- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-94
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-118
- “Constraint Specification Using a Portfolio Object” on page 3-19
- “Portfolio Analysis with Turnover Constraints” on page 4-204
- “Asset Allocation Case Study” on page 4-172
- “Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
- “Portfolio Analysis with Turnover Constraints” on page 4-204
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-183
- “Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215
- “Portfolio Optimization Using Factor Models” on page 4-224
- “Portfolio Optimization Using Social Performance Measure” on page 4-257
- “Diversify Portfolios Using Custom Objective” on page 4-329

More About

- “Portfolio Object” on page 4-19
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-17
- “Setting Up a Tracking Portfolio” on page 4-39

External Websites

- Getting Started with Portfolio Optimization (4 min 13 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 4-13) that enforce upper bounds on net purchases or net sales. One-way turnover constraints can be set using the `Portfolio` object 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` object. 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 4-36).

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` object or `setOneWayTurnover` with empty inputs for the properties to be cleared.

See Also

`Portfolio` | `setDefaultConstraints` | `setBounds` | `setBudget` | `setGroups` | `setGroupRatio` | `setEquality` | `setInequality` | `setTurnover` | `setOneWayTurnover` | `setTrackingPort` | `setTrackingError`

Related Examples

- “Creating the Portfolio Object” on page 4-24
- “Working with Portfolio Constraints Using Defaults” on page 4-57
- “Validate the Portfolio Problem for Portfolio Object” on page 4-90
- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-94
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-118
- “Constraint Specification Using a Portfolio Object” on page 3-19
- “Asset Allocation Case Study” on page 4-172
- “Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-183
- “Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215
- “Portfolio Optimization Using Factor Models” on page 4-224
- “Portfolio Optimization Using Social Performance Measure” on page 4-257
- “Diversify Portfolios Using Custom Objective” on page 4-329

More About

- “Portfolio Object” on page 4-19
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-17
- “Setting Up a Tracking Portfolio” on page 4-39

External Websites

- Getting Started with Portfolio Optimization (4 min 13 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 4-14) that measure the risk relative to a portfolio called a tracking portfolio. Tracking error constraints can be set using the `Portfolio` object 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 4-14.

The tracking error constraint can be set using the `Portfolio` object 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` object. 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 4-39).

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
```

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 4-39).

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 object or setTrackingError with empty inputs for the properties to be cleared.

See Also

`Portfolio` | `setDefaultConstraints` | `setBounds` | `setBudget` | `setGroups` | `setGroupRatio` | `setEquality` | `setInequality` | `setTurnover` | `setOneWayTurnover` | `setTrackingPort` | `setTrackingError`

Related Examples

- “Creating the Portfolio Object” on page 4-24
- “Working with Portfolio Constraints Using Defaults” on page 4-57
- “Validate the Portfolio Problem for Portfolio Object” on page 4-90
- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-94
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-118
- “Constraint Specification Using a Portfolio Object” on page 3-19
- “Asset Allocation Case Study” on page 4-172
- “Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-183
- “Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215
- “Portfolio Optimization Using Factor Models” on page 4-224
- “Portfolio Optimization Using Social Performance Measure” on page 4-257
- “Diversify Portfolios Using Custom Objective” on page 4-329

More About

- “Portfolio Object” on page 4-19
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-17
- “Setting Up a Tracking Portfolio” on page 4-39

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Validate the Portfolio Problem for Portfolio Object

In this section...
"Validating a Portfolio Set" on page 4-90
"Validating Portfolios" on page 4-91

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 4-17.

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-08 *

   -0.3712
       Inf


isbounded =

  logical

   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 =

  logical

   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` | `estimateBounds` | `checkFeasibility`

Related Examples

- “Creating the Portfolio Object” on page 4-24
- “Working with Portfolio Constraints Using Defaults” on page 4-57
- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-94
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-118

- “Asset Allocation Case Study” on page 4-172
- “Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-183
- “Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215
- “Portfolio Optimization Using Factor Models” on page 4-224
- “Portfolio Optimization Using Social Performance Measure” on page 4-257
- “Diversify Portfolios Using Custom Objective” on page 4-329

More About

- “Portfolio Object” on page 4-19
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-17

External Websites

- Getting Started with Portfolio Optimization (4 min 13 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 and this depends on what is your goal:

- To estimate efficient portfolios, see “Obtaining Portfolios Along the Entire Efficient Frontier” on page 4-95.
- To estimate efficient frontiers, see “Estimate Efficient Frontiers for Portfolio Object” on page 4-118.

For information on the workflow when using `Portfolio` objects, see “Portfolio Object Workflow” on page 4-17.

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 an input argument 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
    0.0369    0.1289    0.2209    0.3129    0.4049    0.4969    0.4049    0.2314    0.0579
    0.0404    0.0567    0.0730    0.0893    0.1056    0.1219    0.1320    0.1394    0.1468
    0.0336    0.0929    0.1521    0.2113    0.2705    0.3297    0.4630    0.6292    0.7953    1.0
```

If you want only four portfolios, you can use `estimateFrontier` with `NumPorts` specified as `4`.

```
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)
```

```
pwgt = 4×10

    0.8891    0.7215    0.5540    0.3865    0.2190    0.0515         0         0         0
    0.0369    0.1289    0.2209    0.3129    0.4049    0.4969    0.4049    0.2314    0.0579
    0.0404    0.0567    0.0730    0.0893    0.1056    0.1219    0.1320    0.1394    0.1468
    0.0336    0.0929    0.1521    0.2113    0.2705    0.3297    0.4630    0.6292    0.7953    1.0
```

```
display(pbuy)
```

```
pbuy = 4×10

    0.5891    0.4215    0.2540    0.0865         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    0.0521    0.1113    0.1705    0.2297    0.3630    0.5292    0.6953    0.9
```

```
display(psell)
```

```
psell = 4×10

         0         0         0         0    0.0810    0.2485    0.3000    0.3000    0.3000    0.3
    0.2631    0.1711    0.0791         0         0         0         0    0.0686    0.2421    0.3
    0.1596    0.1433    0.1270    0.1107    0.0944    0.0781    0.0680    0.0606    0.0532    0.2
    0.0664    0.0071         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` | `estimateFrontierLimits` | `estimatePortMoments` | `estimateFrontierByReturn` | `estimatePortReturn` | `estimateFrontierByRisk` | `estimatePortRisk` | `estimateFrontierByRisk` | `estimateMaxSharpeRatio` | `setSolver`

Related Examples

- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-94
- “Creating the Portfolio Object” on page 4-24
- “Working with Portfolio Constraints Using Defaults” on page 4-57
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-118
- “Asset Allocation Case Study” on page 4-172
- “Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-183
- “Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215
- “Portfolio Optimization Using Factor Models” on page 4-224
- “Portfolio Optimization Using Social Performance Measure” on page 4-257
- “Diversify Portfolios Using Custom Objective” on page 4-329

More About

- “Portfolio Object” on page 4-19
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-17

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Obtaining Endpoints of the Efficient Frontier

Often when using a `Portfolio` object, 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)
```

```
pwgt = 4×2

    0.8891         0
    0.0369         0
    0.0404         0
    0.0336    1.0000
```

```
display(pbuy)
```

```
pbuy = 4×2

    0.5891         0
         0         0
         0         0
         0    0.9000
```

```
display(psell)
```

```
psell = 4×2

         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` | `estimateFrontierLimits` | `estimatePortMoments` | `estimateFrontierByReturn` | `estimatePortReturn` | `estimateFrontierByRisk` | `estimatePortRisk` | `estimateFrontierByRisk` | `estimateMaxSharpeRatio` | `setSolver`

Related Examples

- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-94
- “Creating the Portfolio Object” on page 4-24
- “Working with Portfolio Constraints Using Defaults” on page 4-57
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-118
- “Asset Allocation Case Study” on page 4-172
- “Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-183
- “Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215
- “Portfolio Optimization Using Factor Models” on page 4-224
- “Portfolio Optimization Using Social Performance Measure” on page 4-257
- “Diversify Portfolios Using Custom Objective” on page 4-329

More About

- “Portfolio Object” on page 4-19

- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-17

External Websites

- Getting Started with Portfolio Optimization (4 min 13 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 4-98 and “Obtaining Portfolio Risks and Returns” on page 4-118).

```
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` | `estimateFrontierLimits` | `estimatePortMoments` | `estimateFrontierByReturn` | `estimatePortReturn` | `estimateFrontierByRisk` | `estimatePortRisk` | `estimateFrontierByRisk` | `estimateMaxSharpeRatio` | `setSolver`

Related Examples

- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-94
- “Creating the Portfolio Object” on page 4-24
- “Working with Portfolio Constraints Using Defaults” on page 4-57

- “Estimate Efficient Frontiers for Portfolio Object” on page 4-118
- “Asset Allocation Case Study” on page 4-172
- “Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-183
- “Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215
- “Portfolio Optimization Using Factor Models” on page 4-224
- “Portfolio Optimization Using Social Performance Measure” on page 4-257
- “Diversify Portfolios Using Custom Objective” on page 4-329

More About

- “Portfolio Object” on page 4-19
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-17

External Websites

- Getting Started with Portfolio Optimization (4 min 13 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 4-98 and “Obtaining Portfolio Risks and Returns” on page 4-118).

```
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` | `estimateFrontierLimits` | `estimatePortMoments` | `estimateFrontierByReturn` | `estimatePortReturn` | `estimatePortRisk` | `estimateFrontierByRisk` | `estimateMaxSharpeRatio` | `setSolver`

Related Examples

- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-94
- “Creating the Portfolio Object” on page 4-24
- “Working with Portfolio Constraints Using Defaults” on page 4-57
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-118
- “Asset Allocation Case Study” on page 4-172
- “Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-183
- “Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215

- “Portfolio Optimization Using Factor Models” on page 4-224
- “Portfolio Optimization Using Social Performance Measure” on page 4-257
- “Diversify Portfolios Using Custom Objective” on page 4-329

More About

- “Portfolio Object” on page 4-19
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-17

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Efficient Portfolio That Maximizes Sharpe Ratio

Portfolios that maximize the Sharpe ratio are portfolios on the efficient frontier that satisfy several theoretical conditions in finance. Such portfolios are called tangency portfolios since the tangent line from the risk-free rate to the efficient frontier taps the efficient frontier at portfolios that maximize the 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 the portfolio return and risk). For more information, see “Portfolio Optimization Theory” on page 4-3.

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 = 4×1

    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:

```
pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];
p = setInitPort(p, pwgt0);
[pwgt, pbuy, psell] = estimateMaxSharpeRatio(p);

display(pwgt)
```

```
pwgt = 4×1

    0.4251
    0.2917
    0.0856
    0.1977
```

```
display(pbuy)
```

```
pbuy = 4×1

    0.1251
         0
         0
    0.0977
```

```
display(psell)
```

```
psell = 4×1

         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` | `estimateFrontierLimits` | `estimatePortMoments` | `estimateFrontierByReturn` | `estimatePortReturn` | `estimateFrontierByRisk` | `estimatePortRisk` | `estimateFrontierByRisk` | `estimateMaxSharpeRatio` | `setSolver`

Related Examples

- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-94
- “Creating the Portfolio Object” on page 4-24
- “Working with Portfolio Constraints Using Defaults” on page 4-57
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-118
- “Asset Allocation Case Study” on page 4-172
- “Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-183
- “Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215
- “Portfolio Optimization Using Factor Models” on page 4-224
- “Portfolio Optimization Using Social Performance Measure” on page 4-257
- “Diversify Portfolios Using Custom Objective” on page 4-329

More About

- “Portfolio Object” on page 4-19
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-17

External Websites

- Getting Started with Portfolio Optimization (4 min 13 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. Like Optimization Toolbox, which uses the `interior-point-convex` algorithm as the default algorithm for `quadprog`, the portfolio optimization tools also use the `interior-point-convex` algorithm as the default. For details about `quadprog` and quadratic programming algorithms and options, see “Quadratic Programming Algorithms”.

Using 'lcprog' and 'quadprog'

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. Because the solver properties are hidden, you cannot set these options using the `Portfolio` object. The default solver is `lcprog` so you do not need to use `setSolver` to specify this solver. To use `quadprog`, you can set the default `interior-point-convex` algorithm of `quadprog` using this code:

```
p = Portfolio;
p = setSolver(p, 'quadprog');
display(p.solverType)
display(p.solverOptions)
```

```
quadprog options:

   Options used by current Algorithm ('interior-point-convex'):
   (Other available algorithms: 'active-set', 'trust-region-reflective')

   Set properties:
              Algorithm: 'interior-point-convex'
                Display: 'off'
    OptimalityTolerance: 1.0000e-12

   Default properties:
    ConstraintTolerance: 1.0000e-08
           LinearSolver: 'auto'
          MaxIterations: 200
          StepTolerance: 1.0000e-12
```

You can switch back to `lcprog` with this code:

```
p = setSolver(p, 'lcprog');
display(p.solverType);
display(p.solverOptions)
```

```
lcprog
     MaxIter: []
    TieBreak: []
      TolPiv: 5.0000e-08
```

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 name-value arguments in the function call. For example, if you intend to use quadprog and want to use the 'trust-region-reflective' algorithm, call setSolver with this code:

```
p = Portfolio;
p = setSolver(p, 'quadprog', 'Algorithm', 'trust-region-reflective');
display(p.solverOptions)
```

```
  quadprog options:

   Options used by current Algorithm ('trust-region-reflective'):
   (Other available algorithms: 'active-set', 'interior-point-convex')

   Set properties:
              Algorithm: 'trust-region-reflective'

   Default properties:
                Display: 'final'
      FunctionTolerance: 'default dependent on problem'
     HessianMultiplyFcn: []
          MaxIterations: 'default dependent on problem'
    OptimalityTolerance: 'default dependent on problem'
          StepTolerance: 2.2204e-14
    SubproblemAlgorithm: 'cg'
               TypicalX: 'ones(numberOfVariables,1)'
```

In addition, if you want to specify any of the options for quadprog that you typically set through optimoptions from Optimization Toolbox, 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
```

Using the Mixed Integer Nonlinear Programming (MINLP) Solver

The mixed integer nonlinear programming (MINLP) solver, configured using setSolverMINLP, enables you to specify associated solver options for portfolio optimization for a Portfolio object. The MINLP solver is used when any one, or any combination of 'Conditional' BoundType, MinNumAssets, or MaxNumAssets constraints are active. In this case, you formulate the portfolio problem by adding NumAssets binary variables, where 0 indicates not invested, and 1 is invested. For more information on using 'Conditional' BoundType, see setBounds. For more information on specifying MinNumAssets and MaxNumAssets, see setMinMaxNumAssets.

When using the estimate functions with a Portfolio object where 'Conditional' BoundType, MinNumAssets, or MaxNumAssets constraints are active, the mixed integer nonlinear programming (MINLP) solver is automatically used.

Solver Guidelines for Portfolio Objects

The following table provides guidelines for using setSolver and setSolverMINLP.

Portfolio Problem	Portfolio Function	Type of Optimization Problem	Main Solver	Helper Solver
Portfolio without tracking error constraints	`estimateFrontierByRisk`	Optimizing a portfolio for a certain risk level introduces a nonlinear constraint. Therefore, this problem has a linear objective with linear and nonlinear constraints.	`'fmincon'` using `setSolver`	For `'min'`: quadratic objective, `'quadprog'` or `'lcprog'` using `setSolver` For `'max'`: linear objective, `'linprog'` or `'lcprog'` using `setSolver`
Portfolio without tracking error constraints	`estimateFrontierByReturn`	Quadratic objective with linear constraints	`'quadprog'` or `'lcprog'` using `setSolver`	For `‘min’`: quadratic objective, `'quadprog'` or `'lcprog'` using `setSolver` For `‘max’`: linear objective, `'linprog'` or `'lcprog'` using `setSolver`
Portfolio without tracking error constraints	`estimateFrontierLimits`	Quadratic or linear objective with linear constraints	For `‘min’`: quadratic objective, `'quadprog'` or `'lcprog'` using `setSolver` For `‘max’`: linear objective, `'linprog'` or `'lcprog'` using `setSolver`	Not applicable
Portfolio without tracking error constraints	`estimateMaxSharpeRatio`	Quadratic objective with linear constraints	`'quadprog'` using `setSolver`	Because `estimateMaxSharpeRatio` internally calls `estimateFrontierLimits`, all solvers needed by `estimateFrontierLimits` are the helper solvers

Portfolio Problem	Portfolio Function	Type of Optimization Problem	Main Solver	Helper Solver
Portfolio with tracking error constraints	`estimateFrontierByRisk`	Linear objective with linear and nonlinear constraints	`'fmincon'` using `setSolver`	Not applicable
Portfolio with tracking error constraints	`estimateFrontierByReturn`	Linear objective with linear and nonlinear constraints	`'fmincon'` using `setSolver`	Not applicable
Portfolio with tracking error constraints	`estimateFrontierLimits`	Quadratic (minimum risk problem) or linear (maximum return problem) objective with linear and nonlinear constraints	`'fmincon'` using `setSolver`	Not applicable
Portfolio with tracking error constraints	`estimateMaxSharpeRatio`	Quadratic objective with linear and nonlinear constraints	`'fmincon'` using `setSolver`	Not applicable
Portfolio with active `'Conditional'` `BoundType`, `MinNumAssets`, and `MaxNumAssets`	`estimateFrontierByRisk`	The problem is formulated by introducing `NumAssets` binary variables to indicate whether the corresponding asset is invested or not. Therefore, it requires a mixed integer nonlinear programming solver. Three types of MINLP solvers are offered, see `setSolverMINLP`.	Mixed integer nonlinear programming solver (MINLP) using `setSolverMINLP`	`'quadprog'` or `'fmincon'` are used when the `estimate` functions reduce the problem into NLP. These two solvers can be configured through `setSolver`.

Portfolio Problem	Portfolio Function	Type of Optimization Problem	Main Solver	Helper Solver
Portfolio with active `'Conditional'` `BoundType`, `MinNumAssets`, and `MaxNumAssets`	`estimateFrontierByReturn`	The problem is formulated by introducing `NumAssets` binary variables to indicate whether the corresponding asset is invested or not. Therefore, it requires a mixed integer nonlinear programming solver. Three types of MINLP solvers are offered, see `setSolverMINLP`.	Mixed integer nonlinear programming solver (MINLP) using `setSolverMINLP`	`'quadprog'` or `'fmincon'` are used when the `estimate` functions reduce the problem into NLP. These two solvers can be configured through `setSolver`.
Portfolio with active `'Conditional'` `BoundType`, `MinNumAssets`, and `MaxNumAssets`	`estimateFrontierLimits`	The problem is formulated by introducing `NumAssets` binary variables to indicate whether the corresponding asset is invested or not. Therefore, it requires a mixed integer nonlinear programming solver. Three types of MINLP solvers are offered, see `setSolverMINLP`.	Mixed integer nonlinear programming solver (MINLP) using `setSolverMINLP`	`'quadprog'` or `'fmincon'` are used when the `estimate` functions reduce the problem into NLP. These two solvers can be configured through `setSolver`.

Portfolio Problem	Portfolio Function	Type of Optimization Problem	Main Solver	Helper Solver
Portfolio with active `'Conditional'` `BoundType`, `MinNumAssets`, and `MaxNumAssets`	`estimateMaxSharpeRatio`	The problem is formulated by introducing `NumAssets` binary variables to indicate whether the corresponding asset is invested or not. Therefore, it requires a mixed integer nonlinear programming solver. Three types of MINLP solvers are offered, see `setSolverMINLP`.	Mixed integer nonlinear programming solver (MINLP) using `setSolverMINLP`	`'quadprog'` or `'fmincon'` are used when the `estimate` functions reduce the problem into NLP. These two solvers can be configured through `setSolver`.

Solver Guidelines for Custom Objective Problems Using Portfolio Objects

The following table provides guidelines for using `setSolver` and `setSolverMINLP` when using `estimateCustomObjectivePortfolio`.

Portfolio Problem	Portfolio Function	Type of Optimization Problem	Main Solver	Helper Solver
Portfolio with custom linear objective without tracking error constraints	`estimateCustomObjectivePortfolio`	Linear objective with linear constraints	`'linprog'` using `setSolver`	Not applicable
Portfolio with custom quadratic objective without tracking error constraints	`estimateCustomObjectivePortfolio`	Quadratic objective with linear constraints	`'quadprog'` using `setSolver`	Not applicable
Portfolio with custom nonlinear objective without tracking error constraints	`estimateCustomObjectivePortfolio`	Nonlinear objective with linear constraints	`'fmincon'` using `setSolver`	Not applicable
Portfolio with custom linear objective with tracking error constraints	`estimateCustomObjectivePortfolio`	Linear objective with linear and nonlinear constraints	`'fmincon'` using `setSolver`	Not applicable

Portfolio Problem	Portfolio Function	Type of Optimization Problem	Main Solver	Helper Solver
Portfolio with custom quadratic objective with tracking error constraints	`estimateCustomObjectivePortfolio`	Quadratic objective with linear and nonlinear constraints	`'fmincon'` using `setSolver`	Not applicable
Portfolio with custom nonlinear objective with tracking error constraints	`estimateCustomObjectivePortfolio`	Nonlinear objective with linear and nonlinear constraints	`'fmincon'` using `setSolver`	Not applicable
Portfolio with active `'Conditional'` `BoundType`, `MinNumAssets`, or `MaxNumAssets` without tracking error constraints	`estimateCustomObjectivePortfolio`	The problem is formulated by introducing `NumAssets` binary variables to indicate whether the corresponding asset is invested or not. Therefore, it requires a mixed integer nonlinear programming solver. Three types of MINLP solvers are offered, see `setSolverMINLP`.	Mixed integer nonlinear programming solver (MINLP) using `setSolverMINLP`	`'intlinprog'` is used when the problem reduces to a mixed-integer linear problem. This solver can be configured using `setSolverMINLP` and its name-value argument `IntMainSolverOptions`.

See Also

`Portfolio` | `estimatePortReturn` | `estimatePortMoments` | `plotFrontier`

Related Examples

- "Plotting the Efficient Frontier for a Portfolio Object" on page 4-121
- "Creating the Portfolio Object" on page 4-24
- "Working with Portfolio Constraints Using Defaults" on page 4-57
- "Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object" on page 4-94
- "Postprocessing Results to Set Up Tradable Portfolios" on page 4-126
- "Asset Allocation Case Study" on page 4-172
- "Portfolio Optimization Examples Using Financial Toolbox™" on page 4-152
- "Portfolio Optimization with Semicontinuous and Cardinality Constraints" on page 4-183
- "Black-Litterman Portfolio Optimization Using Financial Toolbox™" on page 4-215
- "Portfolio Optimization Using Factor Models" on page 4-224
- "Portfolio Optimization Using Social Performance Measure" on page 4-257

- “Diversify Portfolios Using Custom Objective” on page 4-329

More About

- “Portfolio Object” on page 4-19
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-17
- “Role of Convexity in Portfolio Problems” on page 4-148

External Websites

- Getting Started with Portfolio Optimization (4 min 13 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 4-94 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 4-17.

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` | `estimatePortReturn` | `estimatePortMoments` | `plotFrontier`

Related Examples

- “Plotting the Efficient Frontier for a Portfolio Object” on page 4-121
- “Creating the Portfolio Object” on page 4-24
- “Working with Portfolio Constraints Using Defaults” on page 4-57
- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-94
- “Postprocessing Results to Set Up Tradable Portfolios” on page 4-126
- “Asset Allocation Case Study” on page 4-172
- “Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-183
- “Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215
- “Portfolio Optimization Using Factor Models” on page 4-224
- “Portfolio Optimization Using Social Performance Measure” on page 4-257
- “Diversify Portfolios Using Custom Objective” on page 4-329

More About

- “Portfolio Object” on page 4-19

- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-17

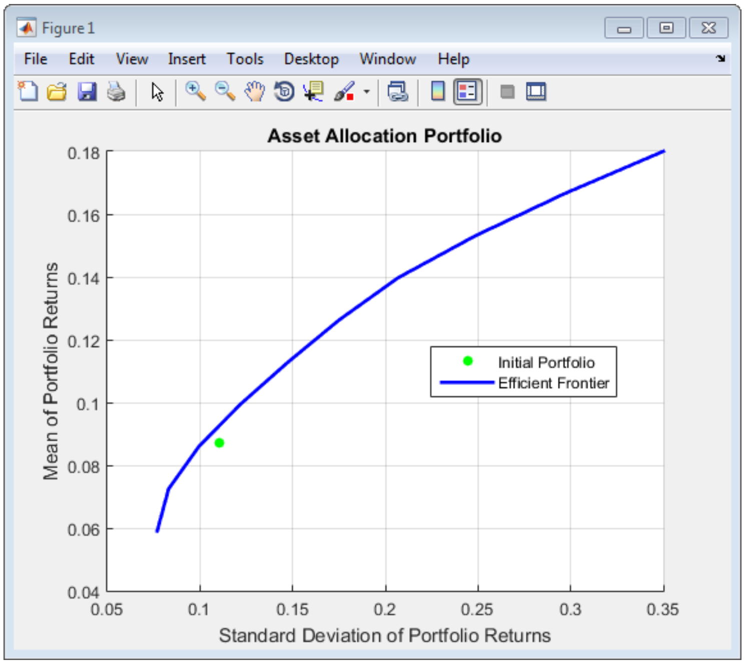
External Websites

- Getting Started with Portfolio Optimization (4 min 13 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 get a plot of the efficient frontier with the default number of portfolios on the frontier (the default number is 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
```

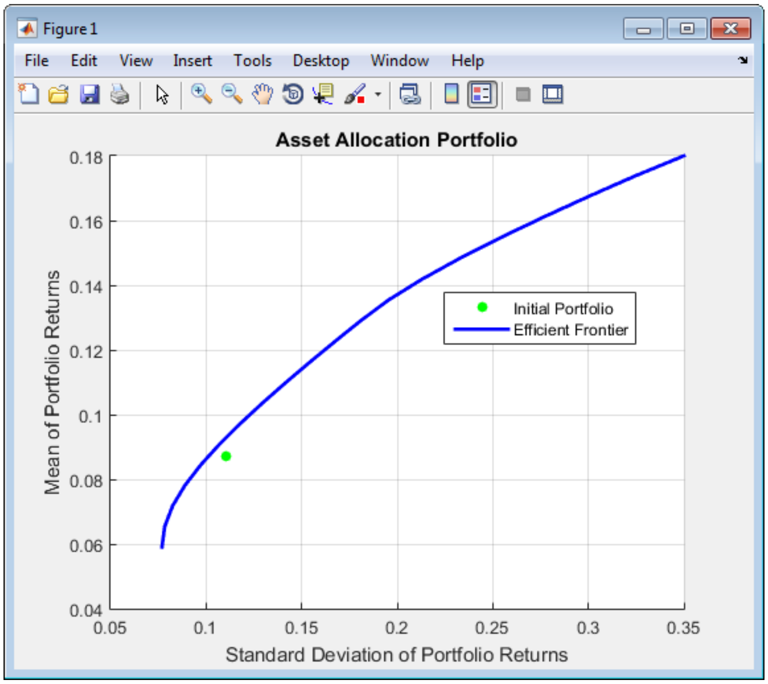
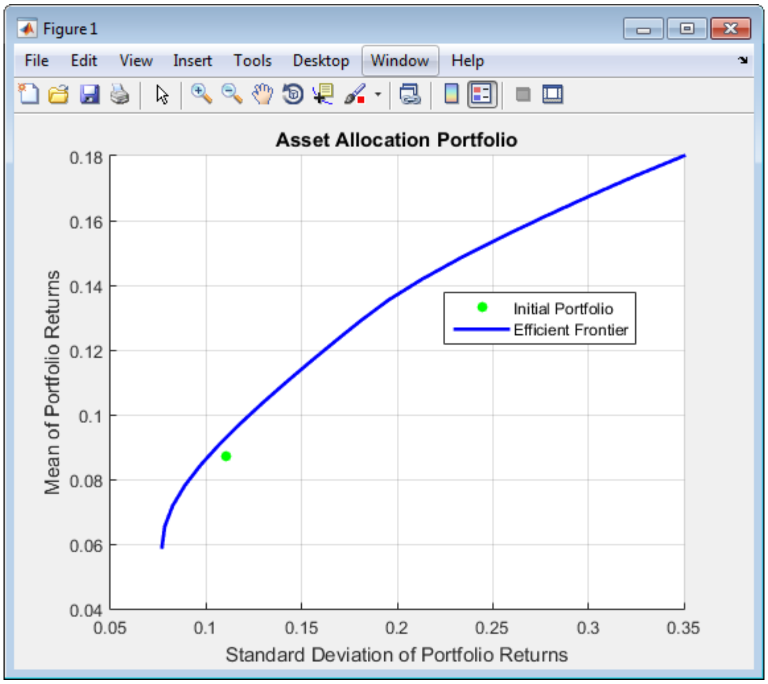


Figure 1
File Edit View Insert Tools Desktop Window Help
Asset Allocation Portfolio
0.18
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0.05
0.1
0.15
0.2
0.25
0.3
0.35
Mean of Portfolio Returns
Standard Deviation of Portfolio Returns
● Initial Portfolio
Efficient Frontier

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 4-94), pass them into `plotFrontier` directly to plot the efficient frontier:

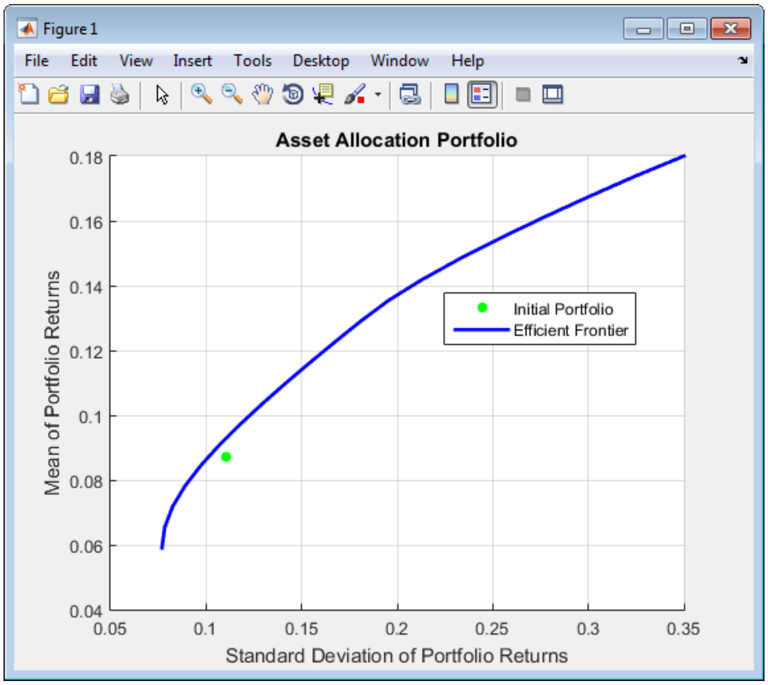
```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
      0.00408 0.0289 0.0204 0.0119;
      0.00192 0.0204 0.0576 0.0336;
      0 0.0119 0.0336 0.1225 ];
pwgt0 = [ 0.3; 0.3; 0.2; 0.1 ];

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` | `estimatePortReturn` | `estimatePortMoments` | `plotFrontier`

Related Examples

- “Estimate Efficient Frontiers for Portfolio Object” on page 4-118
- “Creating the Portfolio Object” on page 4-24
- “Working with Portfolio Constraints Using Defaults” on page 4-57
- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-94
- “Postprocessing Results to Set Up Tradable Portfolios” on page 4-126
- “Asset Allocation Case Study” on page 4-172
- “Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-183
- “Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215
- “Portfolio Optimization Using Factor Models” on page 4-224
- “Portfolio Optimization Using Social Performance Measure” on page 4-257
- “Diversify Portfolios Using Custom Objective” on page 4-329

More About

- “Portfolio Object” on page 4-19
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-17

External Websites

- Getting Started with Portfolio Optimization (4 min 13 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 4-17.

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 4-172.

See Also

`Portfolio` | `estimateAssetMoments` | `checkFeasibility`

Related Examples

- “Troubleshooting Portfolio Optimization Results” on page 4-136
- “Creating the Portfolio Object” on page 4-24
- “Working with Portfolio Constraints Using Defaults” on page 4-57
- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-94
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-118
- “Asset Allocation Case Study” on page 4-172
- “Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-183
- “Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215
- “Portfolio Optimization Using Factor Models” on page 4-224
- “Portfolio Optimization Using Social Performance Measure” on page 4-257
- “Diversify Portfolios Using Custom Objective” on page 4-329

More About

- “Portfolio Object” on page 4-19
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-17

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

When to Use Portfolio Objects Over Optimization Toolbox

While you can use Optimization Toolbox to solve portfolio optimization problems, Financial Toolbox has the `Portfolio`, `PortfolioCVaR`, and `PortfolioMAD` objects that you can use as well. Which tool you use depends on the problem case:

- **Always** use a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object when the problem can most easily be written and implemented using one of these objects. This case includes problems that can be solved only with the internal solvers of the `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. The Optimization Toolbox solvers cannot directly handle these problems. For details, see "Always Use Portfolio, PortfolioCVaR, or PortfolioMAD Object" on page 4-130.
- **Prefer** to use a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object when the problem can be modeled and implemented using both the `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object and the Optimization Toolbox problem-based framework. The advantage of using a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object instead of the Optimization Toolbox is that the internal tools of these objects simplify the analysis. For details, see "Preferred Use of Portfolio, PortfolioCVaR, or PortfolioMAD Object" on page 4-131.
- **Opt** to use Optimization Toolbox for problems that cannot be solved with the internal solvers of the `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. Some of the problems that the `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object cannot solve can be addressed using the Optimization Toolbox problem-based framework. Problems that cannot be directly solved with either framework require either some restructuring of the model or an implementation of a specialized solver. For details, see "Use Optimization Toolbox" on page 4-132.

The following table summarizes the objective functions, constraints, and variables that apply in each case for solving a portfolio problem.

Case for Solving Portfolio Problem	Objective Function	Constraints	Integer (Binary) Variables
"Always" case with Financial Toolbox	• Return — Gross portfolio returns or net portfolio returns • Risk — Variance, CVaR, or MAD • Sharpe ratio (only for mean-variance problems using `Portfolio` object) • Continuous convex functions (only using the `Portfolio` object)	• Return — Gross portfolio returns or net portfolio returns • Risk — Variance, CVaR, or MAD (risk constraints are only supported when the objective is the return) • Linear equalities • Linear inequalities • Tracking error (only for mean-variance problems using `Portfolio` object. Not supported with custom objectives using `estimateCustomObjectivePortfolio`. Tracking error is only supported when the objective is the return, variance, or Sharpe ratio.) • Turnover	• Bounds on the number of assets • Conditional (semicontinuous) bounds (for example, if asset *i* is selected, then $x_i \geq lb_i$, otherwise $x_i = 0$)

Case for Solving Portfolio Problem	Objective Function	Constraints	Integer (Binary) Variables
"Preferred" case with Financial Toolbox	• Return — Gross portfolio returns or net portfolio returns • Risk — Variance, CVaR, or MAD • Sharpe ratio (only for mean-variance problems using `Portfolio` object) • Continuous functions (only using the `Portfolio` object)	• Return — Gross portfolio returns or net portfolio returns • Risk — Variance, CVaR, or MAD (risk constraints are only supported when the objective is the return) • Linear equalities • Linear inequalities • Tracking error (only for mean-variance problems using `Portfolio` object. Not supported with custom objectives using `estimateCustomObjectivePortfolio`. Tracking error is only supported when the objective is the return, variance, or Sharpe ratio.))	None
Optimization Toolbox	Any other nonlinear function not mentioned in “Always Use Portfolio, PortfolioCVaR, or PortfolioMAD Object” on page 4-130 and “Preferred Use of Portfolio, PortfolioCVaR, or PortfolioMAD Object” on page 4-131	Any other nonlinear function not mentioned in “Always Use Portfolio, PortfolioCVaR, or PortfolioMAD Object” on page 4-130 and “Preferred Use of Portfolio, PortfolioCVaR, or PortfolioMAD Object” on page 4-131	None

Always Use Portfolio, PortfolioCVaR, or PortfolioMAD Object

The two general cases for always using the `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object are:

- Problems with *both* supported nonlinear constraints *and* conditional bounds or bounds in the number of assets.

  These problems include:

  - Minimum risk problems subject to constraints for return, linear equality, linear inequality, turnover, and tracking error where the supported risk measures are variance, conditional value-at-risk (CVaR), and mean-absolute-deviation (MAD)
  - Maximum return problems subject to constraints for linear equality, liner inequality, turnover, risk, and tracking error where the supported risk measures are variance, CVaR, and MAD

    Tracking error is supported only for mean-variance problems using the `Portfolio` object. For more information on the supported constraints for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object, see “Portfolio Set for Optimization Using Portfolio Objects” on page 4-8.
  - Custom objective minimization or maximization. The custom objective function (using `estimateCustomObjectivePortfolio`) must be continuous and convex subject to constraints for return, linear equality, linear inequality, and turnover.

  Tracking error is supported using the `Portfolio` object only when the objective is the return, variance or Sharpe ratio. Sharpe ratio is supported only using the `Portfolio` object.

  For more information on the supported constraints and nonlinear risk functions for `Portfolio`, `PortfolioCVaR`, and `PortfolioMAD` objects, see “Portfolio Set for Optimization Using Portfolio Objects” on page 4-8 and “Portfolio Optimization Theory” on page 4-3. The integer (binary) variables can come from either of the following sources: bounds on the number of assets that can be selected in the portfolio or the use of conditional (semicontinuous) bounds for the assets. For example, if asset *i* is selected, then $x_i \geq \texttt{lb}_i$, otherwise $x_i = 0$. These problems cannot be solved using the Optimization Toolbox solvers. However, you can implement your own mixed-integer solver. For more information, see “Mixed-Integer Quadratic Programming Portfolio Optimization: Problem-Based”.
- Problems with turnover constraints and sell or buy costs

  Although the continuous version of these problems can be solved by the Optimization Toolbox solvers, the variable space must be manipulated to rewrite the nonsmooth constraints into equivalent smooth constraints. Given that rewriting the problem requires optimization knowledge, it is recommended to use the `Portfolio`, `PortfolioCVaR`, and `PortfolioMAD` objects instead.

Preferred Use of Portfolio, PortfolioCVaR, or PortfolioMAD Object

The general case for preferred use of the `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object is:

- Continuous problems with minimum risk, maximum return, and maximum Sharpe ratio that are subject to linear equality, liner inequality, turnover, and tracking error constraints.

  Sharpe ratio is supported only for mean-variance problems using the `Portfolio` object. For more information on the supported constraints for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object, see “Portfolio Set for Optimization Using Portfolio Objects” on page 4-8.
- Custom objective minimization or maximization. The custom objective function (using `estimateCustomObjectivePortfolio`) must be continuous and convex subject to constraints for return, linear equality, linear inequality, and turnover.

Tracking error is supported using the `Portfolio` object only when the objective is the return, variance or Sharpe ratio. Sharpe ratio is supported only using the `Portfolio` object.

The supported risk measures are variance, CVaR, and MAD. For more information on the supported constraints for these risk measures, see “Portfolio Set for Optimization Using Portfolio Objects” on page 4-8, “Portfolio Set for Optimization Using PortfolioCVaR Object” on page 5-8, and “Portfolio Set for Optimization Using PortfolioMAD Object” on page 6-7. For all other risk measures and constraints and if tracking error is in the objective, use the Optimization Toolbox.

The advantage of the `Portfolio`, `PortfolioCVaR`, and `PortfolioMAD` object framework over the problem-based framework for the type of problems in the "preferred" case is that common portfolio optimization workflows are leveraged. For example, the `Portfolio`, `PortfolioCVaR`, and `PortfolioMAD` object framework supports the following workflows:

- Estimating and plotting the efficient frontier
- Exchanging the return and risk proxies from the objective function to a constraint
- Solving the maximum Sharpe ratio problem
- Adding bounds on the number of assets selected
- Adding semicontinuous bounds
- Simplifying the use of turnover constraints and sell or buy costs

Use Optimization Toolbox

The two general cases to use Optimization Toolbox is:

- Problems that have nonlinear constraints other than the constraints for risk or tracking error

See Also

`Portfolio` | `PortfolioCVaR` | `PortfolioMAD`

Related Examples

- “Creating the Portfolio Object” on page 4-24
- “Working with Portfolio Constraints Using Defaults” on page 4-57
- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-94
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-118
- “Asset Allocation Case Study” on page 4-172
- “Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-183
- “Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215
- “Portfolio Optimization Using Factor Models” on page 4-224
- “Portfolio Optimization Using Social Performance Measure” on page 4-257
- “Diversify Portfolios Using Custom Objective” on page 4-329

More About

- “Portfolio Object” on page 4-19
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-17
- “Problem-Based Optimization Workflow”

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Comparison of Methods for Covariance Estimation

Mean-variance portfolio optimization is a common technique in portfolio allocation. It usually requires an estimate of the covariance matrix to compute the portfolio weights. Many methods can be used for covariance estimation. The traditional covariance estimate is computed using `cov`. However, in practice the standard covariance estimate is noisy. Since the optimal solution of the portfolio weights is very sensitive to the mean and covariance estimates, noise in the covariance matrix increases the portfolio weights estimation error leading to large turnover and transaction costs. Here are some methods that mitigate the estimation error:

- `robustcov` computes a covariance estimate from a data sample. This function reduces the noise in the estimator by identifying the outliers in the sample and removing them from the final covariance estimate.
- `covarianceShrinkage` computes a covariance estimate from a data sample. This function reduces the noise in the estimator generated from using a finite sample. Shrinkage estimators are used to reduce the mean squared error (MSE) of an estimate. MSE consists of two parts, the bias and the noise. Shrinkage methods reduce the noise in the estimator by pulling the standard covariance estimate towards a target matrix. Thus, the shrinkage estimator results in a biased estimate, but with a lower variance than that of the standard covariance estimate. The method implemented in `covarianceShrinkage` considers a multiple of the identity as the target matrix.

  In covariance shrinkage, the intensity parameter determines how much the standard covariance estimate is pulled towards the target matrix. Because computing the optimal intensity parameter depends on the sample size, `covarianceDenoising` runs slower when the sample size significantly increases.
- `covarianceDenoising` computes a covariance estimate from a data sample or from an initial covariance estimate (usually the standard covariance estimate from `cov`). The `covarianceDenoising` function reduces the noise in the estimator generated from using a finite sample. Unlike `covarianceShrinkage`, the `covarianceDenoising` function differentiates between the noise and the signal in the data to pull only the eigenvalues associated with noise towards a target value. This technique results in an estimate that decreases noise while increasing the signal.

  The computation of the `covarianceDenoising` estimate depends on the number of assets (variables). If the number of assets is too small, there is not enough information to identify between the noisy and meaningful factors. On the other hand, if the number of assets is too large, the underlying optimization problem used to identify the noisy eigenvalues grows, and it takes longer to compute the estimate.

The following table summarizes the information needed to compute the different covariance estimates.

Function	Number of Assets	Sample Size	Sample
`cov` (MATLAB)	Any	Any	Required
`robustcov` (Statistics and Machine Learning Toolbox)	Any	≥ 2 × `numAssets`	Required
`covarianceShrinkage` (Financial Toolbox)	Any (≥ 3 recommended)	Any (≤ 25,000 recommended)	Required

Function	Number of Assets	Sample Size	Sample
`covarianceDenoising` (Financial Toolbox)	Any (≥ 10 recommended)	Any	Not required

Another useful tool for covariance estimation is `nearcorr`. Use `nearcorr` to compute the nearest correlation matrix by minimizing the Frobenius distance to an initial estimate. This is particularly useful to get a positive definite covariance matrix when the number of observations is less than the number of assets.

See Also

`Portfolio` | `covarianceShrinkage` | `covarianceDenoising` | `nearcorr`

Related Examples

- “Creating the Portfolio Object” on page 4-24
- “Compare Performance of Covariance Denoising with Factor Modeling Using Backtesting” on page 4-378
- “Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152

More About

- “Portfolio Object” on page 4-19
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-17

External Websites

- Getting Started with Portfolio Optimization (4 min 13 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` object if you want to modify it, otherwise it creates a new object. See “Creating the Portfolio Object” on page 4-24 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 4-110, 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 faster for smaller problems and the `quadprog` solver is 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 4-22.

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 4-90. 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.

solveContinuousCustomObjProb or solveMICustomObjProb Errors

These errors can occur if the portfolio set is empty or unbounded. If the portfolio set is empty, the error states that the problem is infeasible. The best way to deal with these problems is to use the validation functions in “Validate the Portfolio Problem for Portfolio Object” on page 4-90. Specifically, use `estimateBounds` to examine your portfolio set, and use `checkFeasibility` to ensure that your initial portfolio is either feasible or, if infeasible, that you have sufficient turnover to get from your initial portfolio to the portfolio set.

Efficient Portfolios Do Not Make Sense

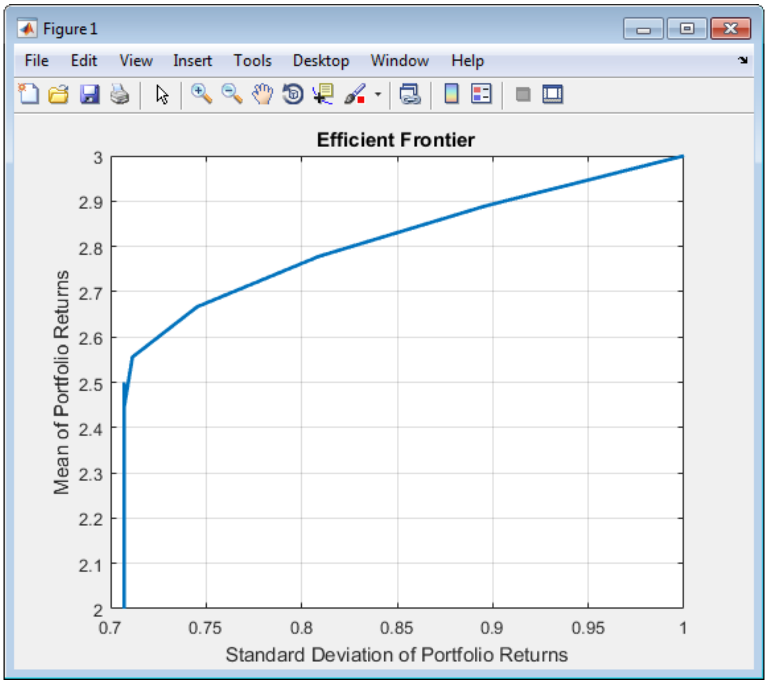
If you obtain efficient portfolios that do not seem to make sense, this can happen if you forget to set specific constraints or you set incorrect constraints. For example, if you allow portfolio weights to fall between `0` and `1` and do not set a budget constraint, you can get portfolios that are 100% 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 rare for standard portfolio problems, but this can occur. For example, this can occur when using unusual combinations of turnover constraints and transaction costs. Usually, 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).

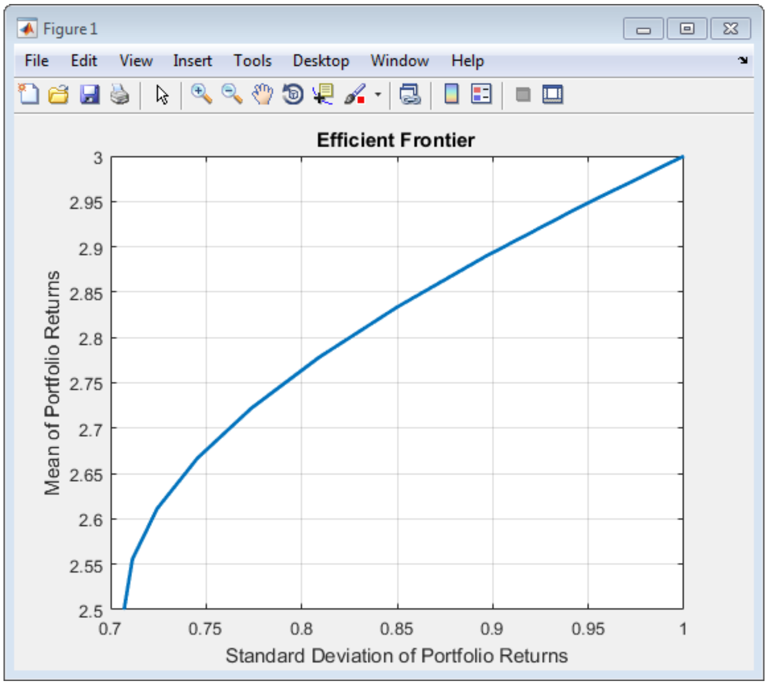
An example of a portfolio optimization problem that has difficulties at the endpoints of the efficient frontier is this standard mean-variance portfolio problem (long-only with a budget constraint) with the following mean and covariance of asset returns:

```
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)
```



Troubleshooting estimateCustomObjectivePortfolio

When using `estimateCustomObjectivePortfolio`, nonconvex functions are not supported for problems with cardinality constraints or conditional bounds. Specifically, the following error displays when using `estimateCustomObjectivePortfolio` with a `Portfolio` object that includes `'Conditional'` `BoundType` (semicontinuous) constraints using `setBounds` or `MinNumAssets` and `MaxNumAssets` (cardinality) constraints using `setMinMaxNumAssets`.

```
Error using solveMICustomObjProb
Objective function must be convex in problems with cardinality constraints and/or conditional bounds.

Error in Portfolio/estimateCustomObjectivePortfolio (line 88)
    [pwgt,exitflag] = solveMICustomObjProb(obj,prob,fun,flags);
```

Note This error applies only to quadratic functions. This error is not detected in nonlinear functions. Therefore, if you are using a nonlinear function, you must validate your input.

For more information, see “Role of Convexity in Portfolio Problems” on page 4-148.

Troubleshooting for Setting 'Conditional' BoundType, MinNumAssets, and MaxNumAssets Constraints

When configuring a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object to include `'Conditional'` `BoundType` (semicontinuous) constraints using `setBounds` or `MinNumAssets` and `MaxNumAssets` (cardinality) constraints using `setMinMaxNumAssets`, the values of the inputs that you supply can result in warning messages.

Conditional Bounds with LowerBound Defined as Empty or Zero

When using `setBounds` with the `BoundType` set to `'Conditional'` and the `LowerBound` input argument is empty (`[ ]`) or `0`, the `Conditional` bound is not effective and is equivalent to a `Simple` bound.

```
AssetMean = [ 0.0101110; 0.0043532; 0.0137058 ];
AssetCovar = [ 0.00324625 0.00022983 0.00420395;
    0.00022983 0.00049937 0.00019247;
    0.00420395 0.00019247 0.00764097 ];
p = Portfolio('AssetMean', AssetMean, 'AssetCovar', AssetCovar, 'Budget', 1);
p = setBounds(p, 0, 0.5, 'BoundType', 'Conditional');
p = setMinMaxNumAssets(p, 3, 3);
estimateFrontier(p, 10)
```

```
Warning: Conditional bounds with 'LowerBound' as zero are equivalent to simple bounds.
Consider either using strictly positive 'LowerBound' or 'simple' as the 'BoundType'
instead.
> In internal.finance.PortfolioMixedInteger/checkBoundType (line 46)
  In Portfolio/checkarguments (line 204)
  In Portfolio/setBounds (line 80)
Warning: The solution may have less than 'MinNumAssets' assets with nonzero weight. To
enforce 'MinNumAssets' requirement, set strictly positive lower conditional bounds.
> In internal.finance.PortfolioMixedInteger/hasIntegerConstraints (line 44)
  In Portfolio/estimateFrontier (line 51)

ans =

  Columns 1 through 8

    0.5000    0.3555    0.3011    0.3299    0.3585    0.3873    0.4160    0.4448
    0.5000    0.5000    0.4653    0.3987    0.3322    0.2655    0.1989    0.1323
    0.0000    0.1445    0.2335    0.2714    0.3093    0.3472    0.3850    0.4229

  Columns 9 through 10

    0.4735    0.5000
    0.0657         0
    0.4608    0.5000
```

In all the 10 optimal allocations, there are allocations (the first and last ones) that only have two assets, which is in conflict with the `MinNumAssets` constraint that three assets should be allocated. Also there are two warnings, which actually explain what happens. In this case, the `'Conditional'` bound constraints are defined as `xi = 0` or `0 <= xi <= 0.5`, which are internally modeled as 0*vi*<=xi<=0.5*vi*, where *vi* is `0` or `1`, where `0` indicates not allocated, and `1` indicates allocated. Here, *vi*=`1`, which still allows the asset to have a weight of `0`. In other words, setting `LowerBound` as `0` or empty, doesn't clearly define the minimum allocation for an allocated asset. Therefore, a `0` weighted asset is also considered as an allocated asset. To fix this warning, follow the instructions in the warning message, and set a `LowerBound` value that is strictly positive.

```
AssetMean = [ 0.0101110; 0.0043532; 0.0137058 ];
AssetCovar = [ 0.00324625 0.00022983 0.00420395;
    0.00022983 0.00049937 0.00019247;
    0.00420395 0.00019247 0.00764097 ];
p = Portfolio('AssetMean', AssetMean, 'AssetCovar', AssetCovar, 'Budget', 1);
p = setBounds(p, 0.3, 0.5, 'BoundType', 'Conditional');
p = setMinMaxNumAssets(p, 3, 3);
estimateFrontier(p, 10)
```

```
ans =

  Columns 1 through 8

    0.3000    0.3180    0.3353    0.3489    0.3580    0.3638    0.3694    0.3576
    0.4000    0.3820    0.3642    0.3479    0.3333    0.3199    0.3067    0.3001
    0.3000    0.3000    0.3005    0.3032    0.3088    0.3163    0.3240    0.3423

  Columns 9 through 10

    0.3289    0.3000
    0.3000    0.3000
    0.3711    0.4000
```

Length of 'BoundType' Must Be Conformable with NumAssets

The `setBounds` optional name-value argument for `'BoundType'` must be defined for all assets in a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. By default, the `'BoundType'` is `'Simple'` and applies to all assets. Using `setBounds`, you can choose to define a `'BoundType'` for each asset. In this case, the number of `'BoundType'` specifications must match the number of assets (`NumAssets`) in the `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. The following example demonstrates the error when the number of `'BoundType'` specifications do not match the number of assets in the `Portfolio` object.

```
AssetMean = [ 0.0101110; 0.0043532; 0.0137058 ];
AssetCovar = [ 0.00324625 0.00022983 0.00420395;
    0.00022983 0.00049937 0.00019247;
    0.00420395 0.00019247 0.00764097 ];
p = Portfolio('AssetMean', AssetMean, 'AssetCovar', AssetCovar, 'Budget', 1);
p = setBounds(p, 0.1, 0.5, 'BoundType',["simple"; "conditional"])
```

```
Cannot create bound constraints.

Caused by:
    Error using internal.finance.PortfolioMixedInteger/checkBoundType (line 28)
    Length of 'BoundType' must be conformable with 'NumAssets'=3.
```

To correct this, modify the `BoundType` to include three specifications because the `Portfolio` object has three assets.

```
AssetMean = [ 0.0101110; 0.0043532; 0.0137058 ];
AssetCovar = [ 0.00324625 0.00022983 0.00420395;
    0.00022983 0.00049937 0.00019247;
    0.00420395 0.00019247 0.00764097 ];

p = Portfolio('AssetMean', AssetMean, 'AssetCovar', AssetCovar, 'Budget', 1);
p = setBounds(p, 0.1, 0.5, 'BoundType',["simple"; "conditional";"conditional"])
p.BoundType
```

```
p = 

  Portfolio with properties:

          BuyCost: []
         SellCost: []
     RiskFreeRate: []
        AssetMean: [3×1 double]
       AssetCovar: [3×3 double]
    TrackingError: []
     TrackingPort: []
         Turnover: []
      BuyTurnover: []
     SellTurnover: []
             Name: []
        NumAssets: 3
        AssetList: []
         InitPort: []
      AInequality: []
      bInequality: []
        AEquality: []
        bEquality: []
       LowerBound: [3×1 double]
       UpperBound: [3×1 double]
      LowerBudget: 1
      UpperBudget: 1
      GroupMatrix: []
       LowerGroup: []
       UpperGroup: []
           GroupA: []
           GroupB: []
       LowerRatio: []
       UpperRatio: []
        BoundType: [3×1 categorical]
     MinNumAssets: []
     MaxNumAssets: []

ans = 

  3×1 categorical array
```

```
    simple
    conditional
    conditional
```

Redundant Constraints from 'BoundType', 'MinNumAssets', 'MaxNumAssets' Constraints

When none of the constraints from `'BoundType'`, `'MinNumAssets'`, or `'MaxNumAssets'` are active, the redundant constraints from `'BoundType'`, `'MinNumAssets'`, `'MaxNumAssets'` warning occurs. This happens when you explicitly use `setBounds` and `setMinMaxNumAssets` but with values that are inactive. That is, the `'Conditional'` `BoundType` has a `LowerBound = [ ]` or `0`, `'MinNumAssets'` is `0`, or `'MaxNumAssets'` is the same value as `NumAssets`. In other words, if any of these three are active, the warning will not show up when using the `estimate` functions or `plotFrontier`. The following two examples show the rationale.

The first example is when the `BoundType` is explicitly set as `'Conditional'` but the `LowerBound` is `0`, and no `'MinNumAssets'` and `'MaxNumAssets'` constraints are defined using `setMinMaxNumAssets`.

```
AssetMean = [ 0.0101110; 0.0043532; 0.0137058 ];
AssetCovar = [ 0.00324625 0.00022983 0.00420395;
    0.00022983 0.00049937 0.00019247;
    0.00420395 0.00019247 0.00764097 ];
p = Portfolio('AssetMean', AssetMean, 'AssetCovar', AssetCovar, 'Budget', 1);
p = setBounds(p, 0, 0.5, 'BoundType', 'Conditional');
estimateFrontier(p, 10)
```

```
Warning: Redundant constraints from 'BoundType', 'MinNumAssets', 'MaxNumAssets'.
> In internal.finance.PortfolioMixedInteger/hasIntegerConstraints (line 24)
  In Portfolio/estimateFrontier (line 51)

ans =

  Columns 1 through 8

    0.5000    0.3555    0.3011    0.3299    0.3586    0.3873    0.4160    0.4448
    0.5000    0.5000    0.4653    0.3987    0.3321    0.2655    0.1989    0.1323
         0    0.1445    0.2335    0.2714    0.3093    0.3471    0.3850    0.4229

  Columns 9 through 10

    0.4735    0.5000
    0.0657         0
    0.4608    0.5000
```

The second example is when you explicitly set the three constraints, but all with inactive values. In this example, the `BoundType` is `'Conditional'` and the `LowerBound` is `0`, thus specifying ineffective `'Conditional'` `BoundType` constraints, and the `'MinNumAssets'` and `'MaxNumAssets'` values are `0` and `3`, respectively. The `setMinMaxNumAssets` function specifies ineffective `'MinNumAssets'` and `'MaxNumAssets'` constraints.

```
AssetMean = [ 0.0101110; 0.0043532; 0.0137058 ];
AssetCovar = [ 0.00324625 0.00022983 0.00420395;
    0.00022983 0.00049937 0.00019247;
    0.00420395 0.00019247 0.00764097 ];
p = Portfolio('AssetMean', AssetMean, 'AssetCovar', AssetCovar, 'Budget', 1);
p = setBounds(p, 0, 0.5, 'BoundType', 'Conditional');
p = setMinMaxNumAssets(p, 0, 3);
estimateFrontier(p, 10)
```

```
Warning: Redundant constraints from 'BoundType', 'MinNumAssets', 'MaxNumAssets'.
> In internal.finance.PortfolioMixedInteger/hasIntegerConstraints (line 24)
  In Portfolio/estimateFrontier (line 51)

ans =

  Columns 1 through 8

    0.5000    0.3555    0.3011    0.3299    0.3586    0.3873    0.4160    0.4448
    0.5000    0.5000    0.4653    0.3987    0.3321    0.2655    0.1989    0.1323
         0    0.1445    0.2335    0.2714    0.3093    0.3471    0.3850    0.4229

  Columns 9 through 10
```

```
    0.4735    0.5000
    0.0657         0
    0.4608    0.5000
```

Infeasible Portfolio Problem with 'BoundType', 'MinNumAssets', 'MaxNumAssets'

The `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object performs validations of all the constraints that you set before solving any specific optimization problems. The `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object first considers all constraints other than `'Conditional'` `BoundType`, `'MinNumAssets'`, and `'MaxNumAssets'` and issues an error message if they are not compatible. Then the `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object adds the three constraints to check if they are compatible with the already checked constraints. This separation is natural because `'Conditional'` `BoundType`, `'MinNumAssets'`, and `'MaxNumAssets'` require additional binary variables in the mathematical formulation that leads to a MINLP, while other constraints only need continuous variables. You can follow the error messages to check when the infeasible problem occurs and take actions to fix the constraints.

One possible scenario is when the `BoundType` is `'Conditional'` and Groups are defined for the `Portfolio` object. In this case, the Group definitions are themselves in conflict. Consequently, the `'Conditional'` bound constraint cannot be applied when running `estimateFrontierLimits`.

```
AssetMean = [ 0.0101110; 0.0043532; 0.0137058 ];
AssetCovar = [ 0.00324625 0.00022983 0.00420395;
    0.00022983 0.00049937 0.00019247;
    0.00420395 0.00019247 0.00764097 ];

p = Portfolio('AssetMean', AssetMean, 'AssetCovar', AssetCovar, 'Budget', 1);
p = setBounds(p, 0.1, 0.5, 'BoundType','Conditional');
p = setGroups(p, [1,1,0], 0.3, 0.5);
p = addGroups(p, [0,1,0], 0.6, 0.7);
pwgt = estimateFrontierLimits(p)
```

```
Error using Portfolio/buildMixedIntegerProblem (line 31)
Infeasible portfolio problem prior to considering 'BoundType', 'MinNumAssets',
'MaxNumAssets'. Verify if constraints from groups, bounds, group ratios, inequality,
equality, etc. are compatible.

Error in Portfolio/estimateFrontierLimits>int_frontierLimits (line 93)
ProbStruct = buildMixedIntegerProblem(obj);

Error in Portfolio/estimateFrontierLimits (line 73)
    pwgt = int_frontierLimits(obj, minsolution, maxsolution);
```

To correct this error, change the `LowerGroup` in the `addGroups` function to also be `0.3` to match the `GroupMatrix` input from `setGroups`.

```
AssetMean = [ 0.0101110; 0.0043532; 0.0137058 ];
AssetCovar = [ 0.00324625 0.00022983 0.00420395;
    0.00022983 0.00049937 0.00019247;
    0.00420395 0.00019247 0.00764097 ];

p = Portfolio('AssetMean', AssetMean, 'AssetCovar', AssetCovar, 'Budget', 1);
p = setBounds(p, 0.1, 0.5, 'BoundType','Conditional');
p = setGroups(p, [1,1,0], 0.3, 0.5);
p = addGroups(p, [0,1,0], 0.3, 0.7);
pwgt = estimateFrontierLimits(p)
```

```
pwgt =

         0    0.2000
    0.5000    0.3000
    0.5000    0.5000
```

A second possible scenario is when the `BoundType` is `'Conditional'` and the `setEquality` function is used with the `bEquality` parameter set to `0.04`. This sets an equality constraint to have *x*1 + *x*3 = `0.04`. At the same time, `setBounds` also set the semicontinuous constraints to have *xi* = `0` or `0.1` <= *xi* <= `2.5`, which lead to *x*1 + *x*3 = `0` or `0.1` <= *x*1 + *x*3 <= `5`. The semicontinuous constraints are not compatible with the equality constraint because there is no way to get x1 + x3 to equal 0.04. Therefore, the error message is displayed.

```
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('AssetMean', AssetMean, 'AssetCovar', AssetCovar, 'Budget', 1);
A = [ 1 0 1 0 ];
b = 0.04;
p = setEquality(p, A, b);
p = setBounds(p, 0.1, 2.5, 'BoundType','Conditional');
p = setMinMaxNumAssets(p, 2, 2);
pwgt = estimateFrontierLimits(p)
```

```
Error using Portfolio/buildMixedIntegerProblem (line 109)
Infeasible portfolio problem when considering 'BoundType', 'MinNumAssets',
'MaxNumAssets'. Verify if these are compatible with constraints from groups, bounds,
group ratios, inequality, equality, etc.

Error in Portfolio/estimateFrontierLimits>int_frontierLimits (line 93)
ProbStruct = buildMixedIntegerProblem(obj);

Error in Portfolio/estimateFrontierLimits (line 73)
    pwgt = int_frontierLimits(obj, minsolution, maxsolution);
```

To correct this error, change the bEquality parameter from 0.04 to .4.

```
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('AssetMean', AssetMean, 'AssetCovar', AssetCovar, 'Budget', 1);
A = [ 1 0 1 0 ];
b = 0.4;
p = setEquality(p, A, b);
p = setBounds(p, 0.1, 2.5, 'BoundType','Conditional');
p = setMinMaxNumAssets(p, 2, 2);
pwgt = estimateFrontierLimits(p)
```

```
pwgt =

    0.4000         0
    0.6000         0
         0    0.4000
         0    0.6000
```

Unbounded Portfolio Problem

This error occurs when you are using a Portfolio, PortfolioCVaR, or PortfolioMAD object and there is no UpperBound defined in setBounds and you are using setMinMaxNumAssets. In this case, this is formulated as a mixed integer programming problem and an UpperBound is required to enforce MinNumAssets and MaxNumAssets constraints.

The optimizer first attempts to estimate the upper bound of each asset, based on all the specified constraints. If the UpperBound cannot be found, an error message occurs which instructs you to set an explicit UpperBound. In most cases, as long as you set some upper bounds to the problem using any set function, the optimizer can successfully find a good estimation.

```
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('AssetMean', AssetMean, 'AssetCovar', AssetCovar);
p = setBounds(p, 0.1, 'BoundType','Conditional');
p = setGroups(p, [1,1,1,0], 0.3, 0.5);
p = setMinMaxNumAssets(p, 3, 3);
pwgt = estimateFrontierLimits(p)
```

```
Error using Portfolio/buildMixedIntegerProblem (line 42)
Unbounded portfolio problem. Upper bounds cannot be inferred from the existing
constraints. Set finite upper bounds using 'setBounds'.

Error in Portfolio/estimateFrontierLimits>int_frontierLimits (line 93)
ProbStruct = buildMixedIntegerProblem(obj);

Error in Portfolio/estimateFrontierLimits (line 73)
    pwgt = int_frontierLimits(obj, minsolution, maxsolution);
```

To correct this error, specify an `UpperBound` value for `setBounds`.

```
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('AssetMean', AssetMean, 'AssetCovar', AssetCovar);
p = setBounds(p, 0.1, .9, 'BoundType','Conditional');
p = setGroups(p, [1,1,1,0], 0.3, 0.5);
p = setMinMaxNumAssets(p, 3, 3);
pwgt = estimateFrontierLimits(p)
```

```
pwgt =

    0.1000         0
    0.1000    0.1000
    0.1000    0.4000
         0    0.9000
```

Total Number of Portfolio Weights with a Value > 0 Are Greater Than MaxNumAssets Specified

When using a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object, the optimal allocation `w` may contain some very small values that leads to `sum(w>0)` larger than `MaxNumAssets`, even though the `MaxNumAssets` constraint is specified using `setMinMaxNumAssets`. For example, in the following code when `setMinMaxNumAssets` is used to set `MaxNumAssets` to `15`, the `sum(w>0)` indicates that there are `19` assets. A close examination of the weights shows that the weights are extremely small and are actually 0.

```
T = readtable('dowPortfolio.xlsx');
symbol = T.Properties.VariableNames(3:end);
assetReturn = tick2ret(T{:,3:end});
p = Portfolio('AssetList', symbol, 'budget', 1);
p = setMinMaxNumAssets(p, 10, 15);
p = estimateAssetMoments(p,assetReturn);
p = setBounds(p,0.01,0.5,'BoundType','Conditional','NumAssets',30);
p = setTrackingError(p,0.05,ones(1, p.NumAssets)/p.NumAssets);
w = estimateFrontierLimits(p,'min'); % minimum risk portfolio

sum(w>0)   % Number of assets that are allocated in the optimal portfolio
w(w<eps)   % Check the weights of the very small weighted assets
```

```
ans =

    19


ans =

   1.0e-20 *

  -0.0000
        0
        0
   0.0293
        0
   0.3626
   0.2494
        0
   0.0926
  -0.0000
        0
   0.0020
        0
```

```
         0
         0
         0
```

This situation only happens when the `OuterApproximation` algorithm is used with `setSolverMINLP` to solve a MINLP portfolio optimization problem. The `OuterApproximation` internally fixes the latest solved integer variables and runs an NLP with `quadprog` or `fmincon`, which introduces numerical issues and leads to weights that are very close to 0.

If you do not want to deal with very small values, you can use `setSolverMINLP` to select a different algorithm. In this example, the `'TrustRegionCP'` algorithm is specified.

```
T = readtable('dowPortfolio.xlsx');
symbol = T.Properties.VariableNames(3:end);
assetReturn = tick2ret(T{:,3:end});
p = Portfolio('AssetList', symbol, 'budget', 1);
p = setMinMaxNumAssets(p, 10, 15);
p = estimateAssetMoments(p,assetReturn);
p = setBounds(p,0.01,0.5,'BoundType','Conditional','NumAssets',30);
p = setTrackingError(p,0.05,ones(1, p.NumAssets)/p.NumAssets);
p = setSolverMINLP(p,'TrustRegionCP');
w = estimateFrontierLimits(p,'min'); % minimum risk portfolio

sum(w>0)   % Number of assets that are allocated in the optimal portfolio
w(w<eps)   % The weights of the very small weighted assets are strictly zeros
```

```
ans =

    14


ans =

     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
```

See Also

`Portfolio` | `estimateAssetMoments` | `checkFeasibility` | `setBounds` | `setMinMaxNumAssets`

Related Examples

- "Postprocessing Results to Set Up Tradable Portfolios" on page 4-126
- "Creating the Portfolio Object" on page 4-24
- "Working with Portfolio Constraints Using Defaults" on page 4-57
- "Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object" on page 4-94
- "Estimate Efficient Frontiers for Portfolio Object" on page 4-118
- "Asset Allocation Case Study" on page 4-172
- "Portfolio Optimization Examples Using Financial Toolbox™" on page 4-152
- "Portfolio Optimization with Semicontinuous and Cardinality Constraints" on page 4-183

- “Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215
- “Portfolio Optimization Using Factor Models” on page 4-224
- “Portfolio Optimization Using Social Performance Measure” on page 4-257
- “Diversify Portfolios Using Custom Objective” on page 4-329
- “Troubleshooting CVaR Portfolio Optimization Results” on page 5-115
- “Troubleshooting MAD Portfolio Optimization Results” on page 6-110

More About

- “Portfolio Object” on page 4-19
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-17

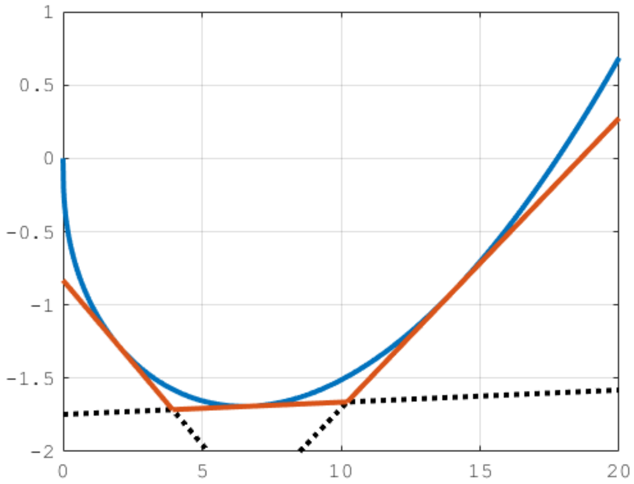
External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

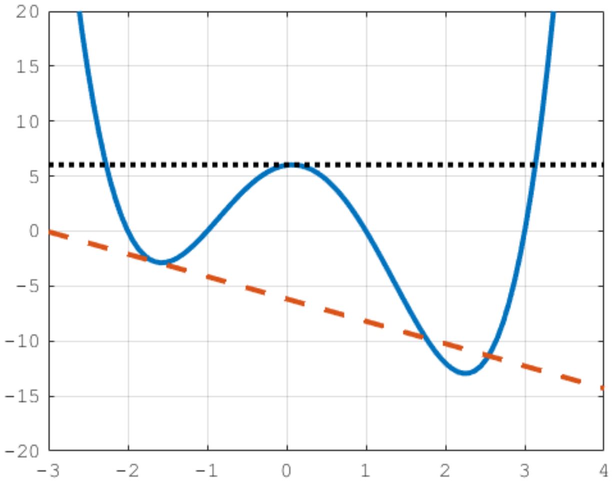
Role of Convexity in Portfolio Problems

When using a `Portfolio` object and the `estimateCustomObjectivePortfolio` function with the Financial Toolbox solvers, the solvers return a local minimum, which is not guaranteed to be a global minimum. A local minimum of a function is a point where the function value is smaller than the points around it, but the value is larger than a distant point. A global minimum is the point where the function value is smaller than at all other points. Convex functions play an important role in optimization because they guarantee that all local minima are global minima. In other words, any solution found by the solver is guaranteed to be a global minimum.

Another important feature of convex functions is that all tangents of the function lie underneath the function. This feature is particularly useful in problems with cardinality constraints or conditional bounds. For more information on cardinality constraints and conditional bounds, see “Working with 'Conditional' BoundType, MinNumAssets, and MaxNumAssets Constraints Using Portfolio Objects” on page 4-78 and `setMinMaxNumAssets`. The Financial Toolbox internal solvers for portfolio problems with binary variables use an iterative method that approximates the nonlinear functions (for example, variance) with a collection of linear functions. The linear functions used to approximate the original function are the tangents of the function at different points. Because the tangents for convex functions always lie underneath the function, the Financial Toolbox solvers guarantee that the approximation does not remove any minimum from the problem.



Using Financial Toolbox solvers for portfolio problems with nonconvex functions and cardinality constraints or conditional bounds might result in suboptimal solutions. The reason is that the tangent lines used to approximate the nonconvex functions might cut off different minima as demonstrated in the following figure:



Even though the red dashed tangent line cuts off only a small piece of the graph, that small piece already contains an important point, the global minima. In the worst case scenario, the Financial Toolbox solvers compute the tangent line at a point that removes all minima of the function, like the black dotted line does. Therefore, it is important to be aware of the type of function passed to a Financial Toolbox solver.

All the above applies to minimization problems. Conversely, maximization problems with cardinality constraints or conditional bounds can be solved by the Financial Toolbox solvers only if the functions are concave.

Examples of Convex Functions

Here are some examples of common convex functions used in finance. You can use these convex functions in minimization problems when using a `Portfolio` object with the `estimateCustomObjectivePortfolio` function:

- Variance:

  $$x^T \Sigma x$$
- Tracking error:

  $$(x - x_0)^T \Sigma (x - x_0)$$
- Herfindahl-Hirschman index:

  $$x^T x$$
- Any linear function
- Any positive combination of convex functions:

  $$x^T \Sigma x + \lambda({}^{x}\Sigma(x - x_0),\ \lambda \geq 0$$

  $$x^T \Sigma x + \lambda_1 x^T x + \lambda_2({}^{x}\Sigma(x - x_0),\ \lambda_1, \lambda_2 \geq 0$$

Note If the portfolio problem has cardinality constraints or conditional bounds specified using `setMinMaxNumAssets` or `setBounds`, the objective function must also be convex.

Examples of Concave Functions

Here are some examples of common concave functions used in finance. You can use concave functions in maximization problems when using a `Portfolio` object with the `estimateCustomObjectivePortfolio` function.

- Risk penalized return:

  $$\mu^T x - \lambda x^T \Sigma x,\ \lambda \geq 0$$
- Any linear function
- Any positive combination of concave functions

Examples of Nonconvex Functions

Here are some examples of common nonconvex functions used in finance. You can use nonconvex functions in minimization or maximization problems when using a `Portfolio` object *without cardinality constraints or conditional bounds* with the `estimateCustomObjectivePortfolio` function.

- Sharpe ratio:

  $$\frac{\mu^T x - r_f}{\sqrt{x^T \Sigma x}}$$
- Information ratio:

  $$\frac{\mu^T (x - x_0)}{\sqrt{(x - x_0)^T \Sigma (x - x_0)}}$$
- Diversification ratio:

  $$\frac{\sigma^T x}{\sqrt{x^T \Sigma x}}$$
- Risk budgeting deviation:

  $$\sum_i \left(\frac{x_i (\Sigma x)_i}{x^T \Sigma x} - b_i \right)^2$$
- The following function is a special case:

  $$x^T \Sigma x - \lambda \sum_i \ln x_i$$

  This function is commonly used to construct risk parity portfolios. Although this function is convex, problems with cardinality constraints or conditional bounds make this function ill-defined. This happens because $\ln x_i$ is not defined when $x_i = 0$. Since adding cardinality constraints or conditional bounds imply that some assets have a weight of 0, the function becomes ill-defined at the points of interest.

Note If the portfolio problem has cardinality constraints or conditional bounds specified using `setMinMaxNumAssets` or `setBounds`, the objective function cannot be nonconvex.

See Also

`Portfolio` | `estimateCustomObjectivePortfolio`

Related Examples

- "Risk Parity or Budgeting with Constraints" on page 4-356
- "Solve Robust Portfolio Maximum Return Problem with Ellipsoidal Uncertainty" on page 4-349
- "Solve Problem for Minimum Tracking Error with Net Return Constraint" on page 4-347
- "Solve Problem for Minimum Variance Portfolio with Tracking Error Penalty" on page 4-342
- "Portfolio Optimization Against a Benchmark" on page 4-195
- "Portfolio Optimization Using Social Performance Measure" on page 4-257
- "Diversify Portfolios Using Custom Objective" on page 4-329

More About

- "Solver Guidelines for Custom Objective Problems Using Portfolio Objects" on page 4-115

Portfolio Optimization Examples Using Financial Toolbox™

Follow a sequence of examples that highlight features of the `Portfolio` object. Specifically, the examples use the `Portfolio` object to 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

First create a "standard" `Portfolio` object with `Portfolio` to incorporate the list of assets, the risk-free rate, and the moments of asset returns into the object.

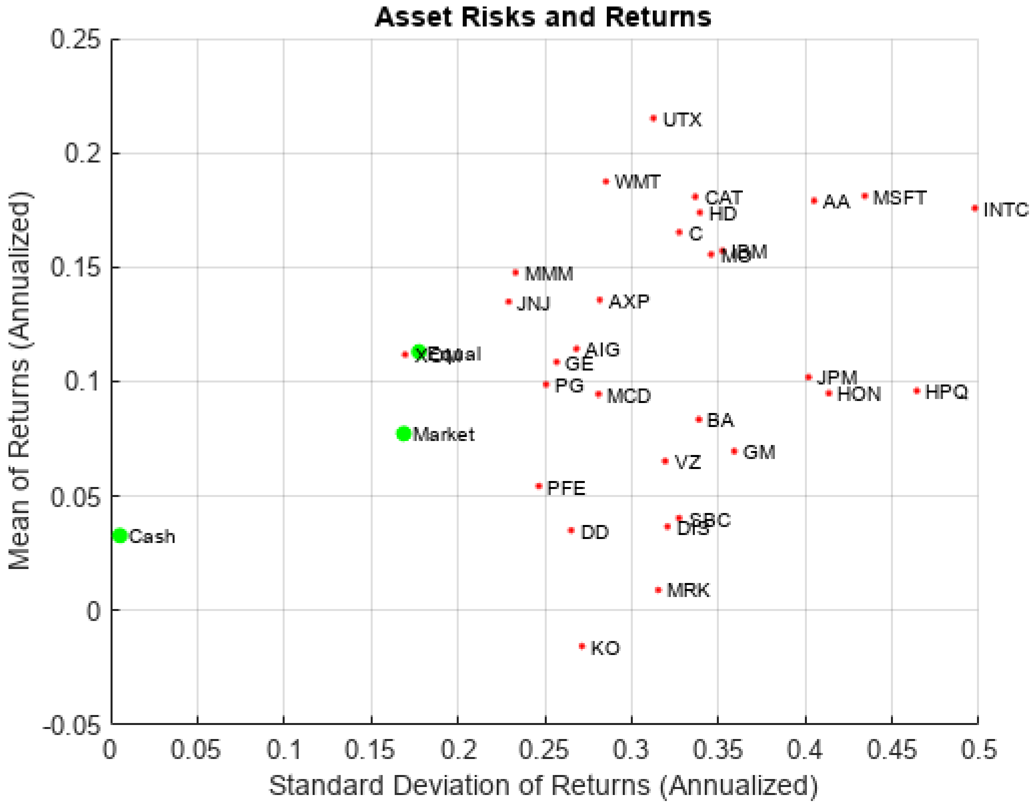
```
p = Portfolio('AssetList',AssetList,'RiskFreeRate',CashMean);
p = setAssetMoments(p,AssetMean,AssetCovar);
```

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. The 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 `portfolioexamples_plot` function converts monthly total returns into annualized total returns.

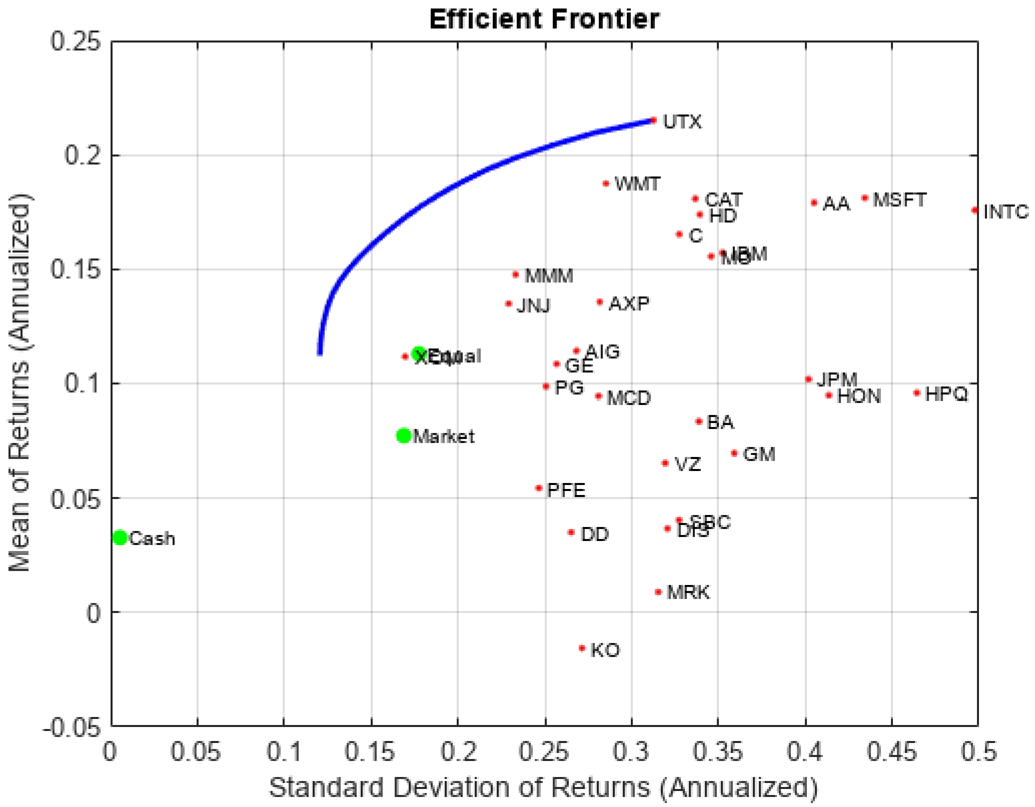
```
clf;
portfolioexamples_plot('Asset Risks and Returns', ...
    {'scatter', mrsk, mret, {'Market'}}, ...
    {'scatter', crsk, cret, {'Cash'}}, ...
    {'scatter', ersk, eret, {'Equal'}}, ...
    {'scatter', sqrt(diag(p.AssetCovar)), p.AssetMean, p.AssetList, '.r'});
```



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 the 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'});
```

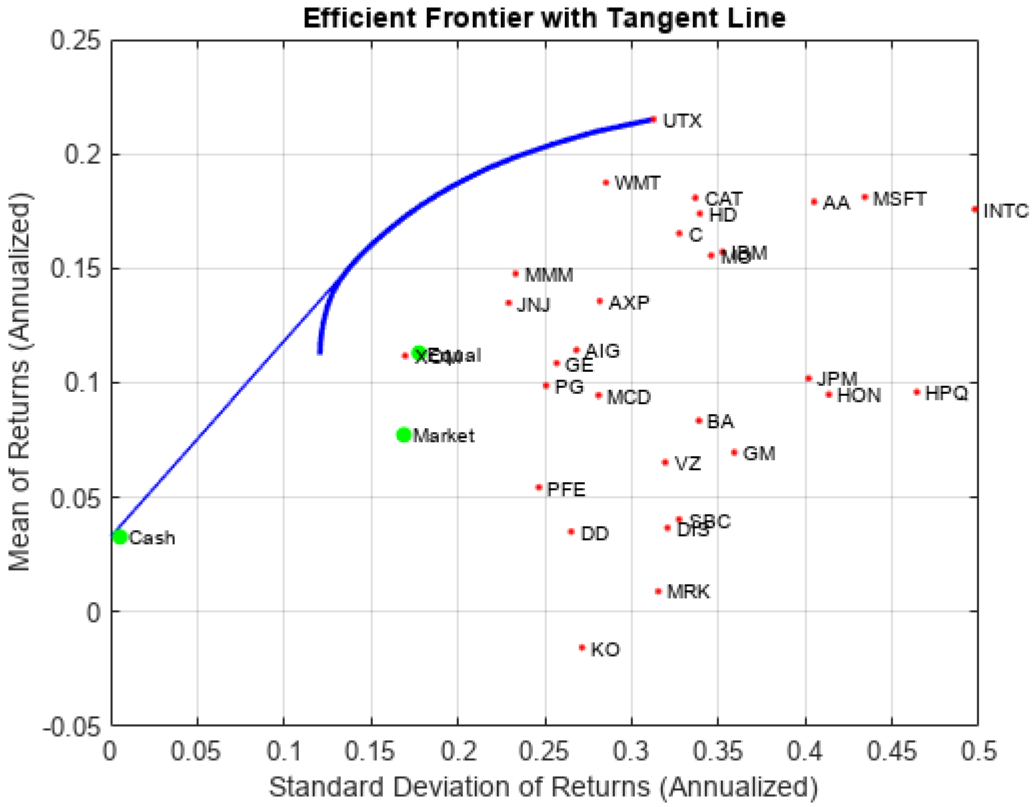


Illustrate the Tangent Line to the Efficient Frontier

Tobin's mutual fund theorem (Tobin 1958 on page 4-170) says that the portfolio allocation problem is viewed as a decision to allocate between a riskless asset and a risky portfolio. In the mean-variance framework, cash serves 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 `Portfolio` or the "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 is accomplished with the `estimateFrontierLimits` function.

```
[rsk,ret] = estimatePortMoments(p,estimateFrontierLimits(p));
```

```
display(rsk)
```

```
rsk = 2×1

    0.0348
    0.0903
```

```
display(ret)
```

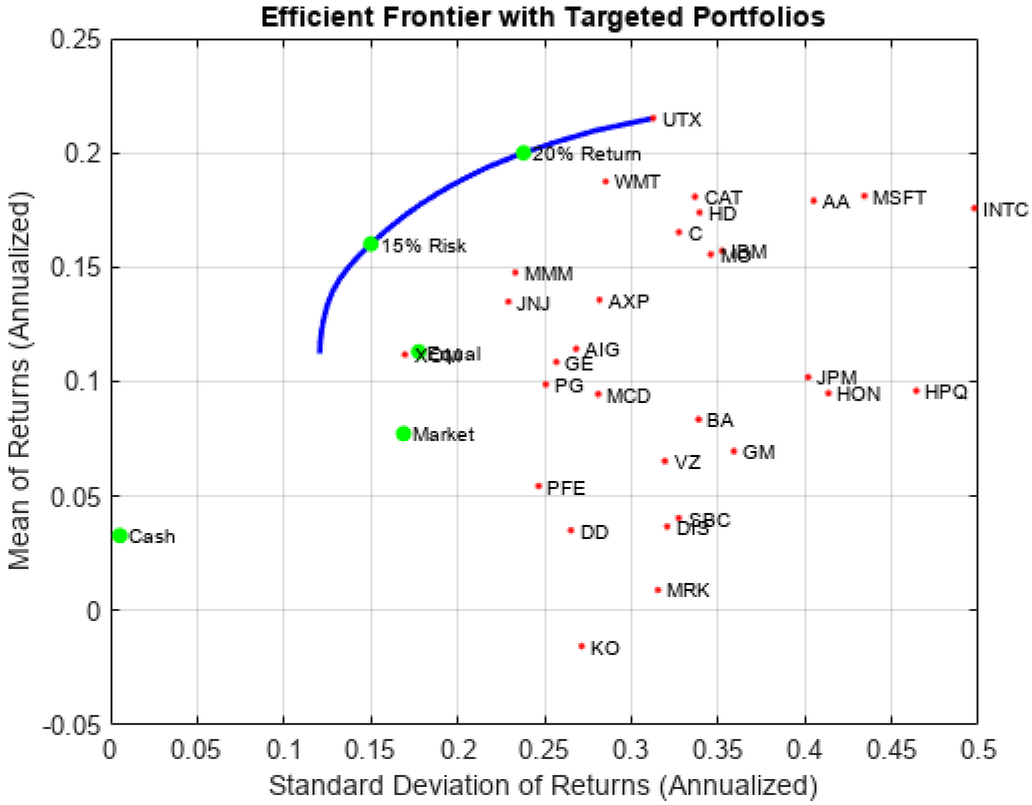
```
ret = 2×1

    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), aBlotter, false
```

```
Portfolio with 20% Target Return
            Weight
    CAT      1.1445
    INTC    0.17452
    MO       9.6521
    MSFT    0.85862
    UTX      56.918
    WMT      31.253
```

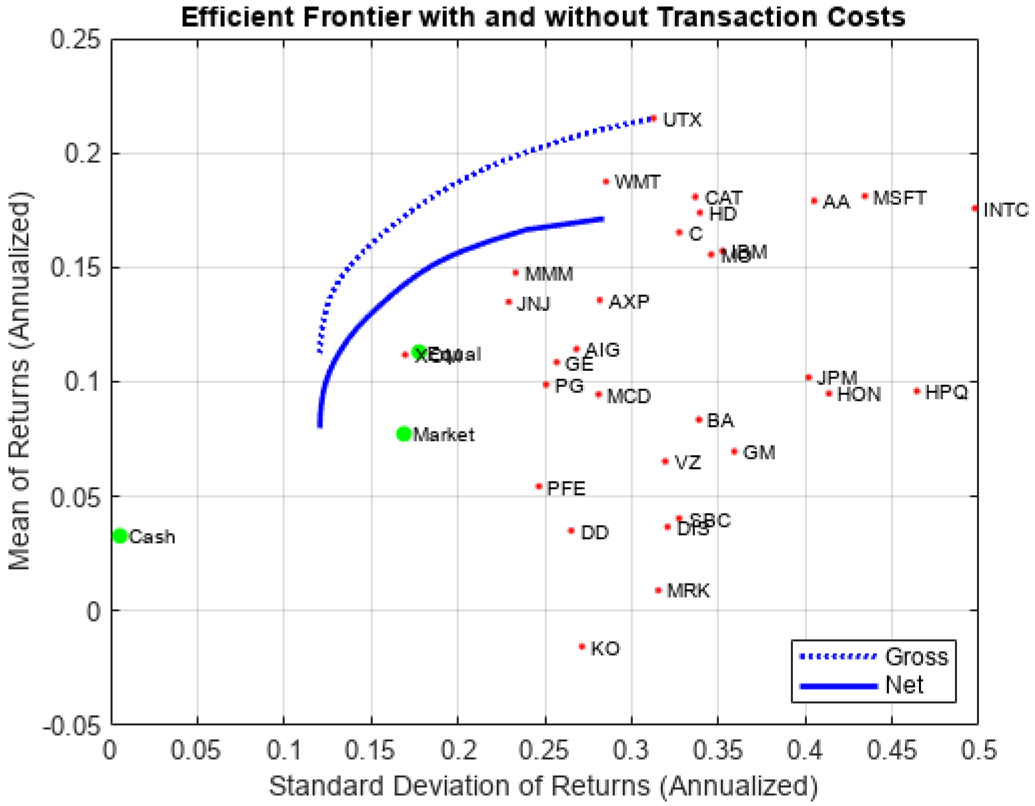
```
bBlotter = dataset({100*bwgt(bwgt > 0),'Weight'}, 'obsnames', p.AssetList(bwgt > 0));

displayPortfolio(sprintf('Portfolio with %g%% Target Risk', 100*TargetRisk), bBlotter, false);
```

```
Portfolio with 15% Target Risk
            Weight
    AA      3.1928e-22
    AIG     5.5874e-21
    AXP     4.3836e-21
    BA       6.257e-22
    C       8.7778e-21
    GE      4.3302e-22
    GM      5.3508e-22
    HD      9.4473e-21
    HON     2.2103e-22
    IBM     3.8403e-20
    INTC        2.2585
    JNJ         9.2162
    MCD     3.7312e-22
    MMM         16.603
    MO          15.388
    MRK      1.641e-21
    MSFT        4.4467
    PFE     1.5991e-21
    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.

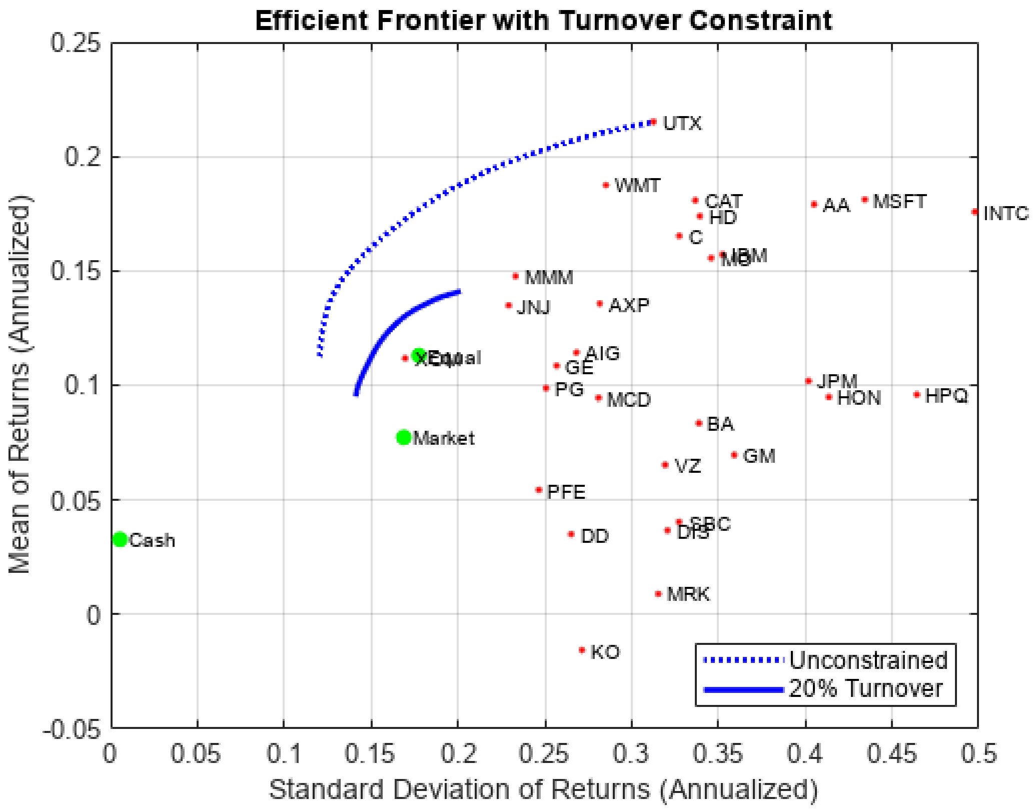
```
BuyCost = 0.0020;
SellCost = 0.0020;

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 is 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)
```

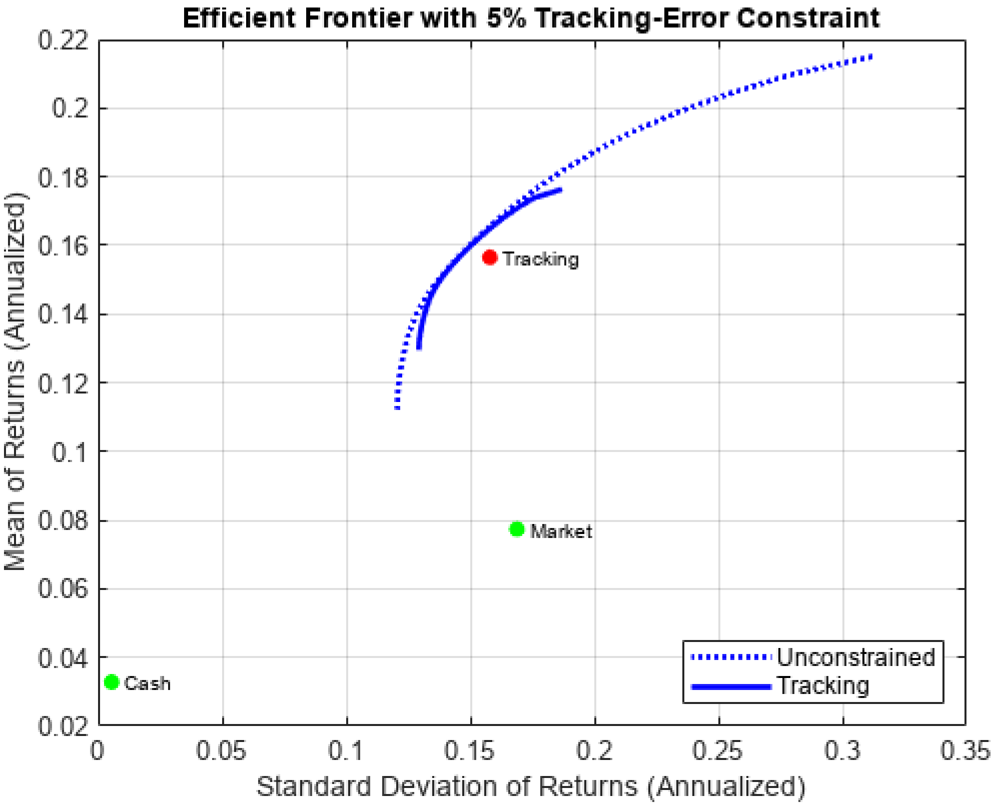
```
Sum of Purchases by Portfolio along Efficient Frontier (Max. Turnover 20%)
20.0000 20.0000 20.0000 20.0000 20.0000 20.0000 20.0000 20.0000 20.0000 20.0000 20.0000 20.0000
Sum of Sales by Portfolio along Efficient Frontier (Max. Turnover 20%)
20.0000 20.0000 20.0000 20.0000 20.0000 20.0000 20.0000 20.0000 20.0000 20.0000 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 the tracking por

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 the 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'});
```

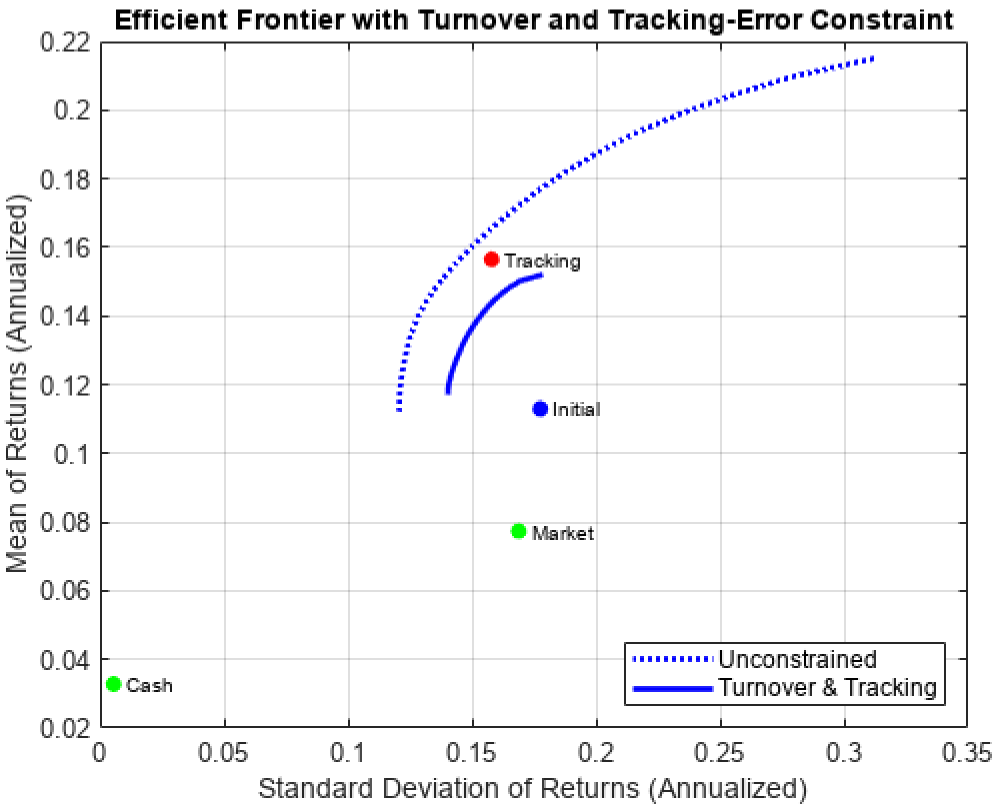


Combined Turnover and Tracking-Error Constraints

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 a maximum of 30% turnover and the tracking-error constraint has a maximum of 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 tracking portfol
```

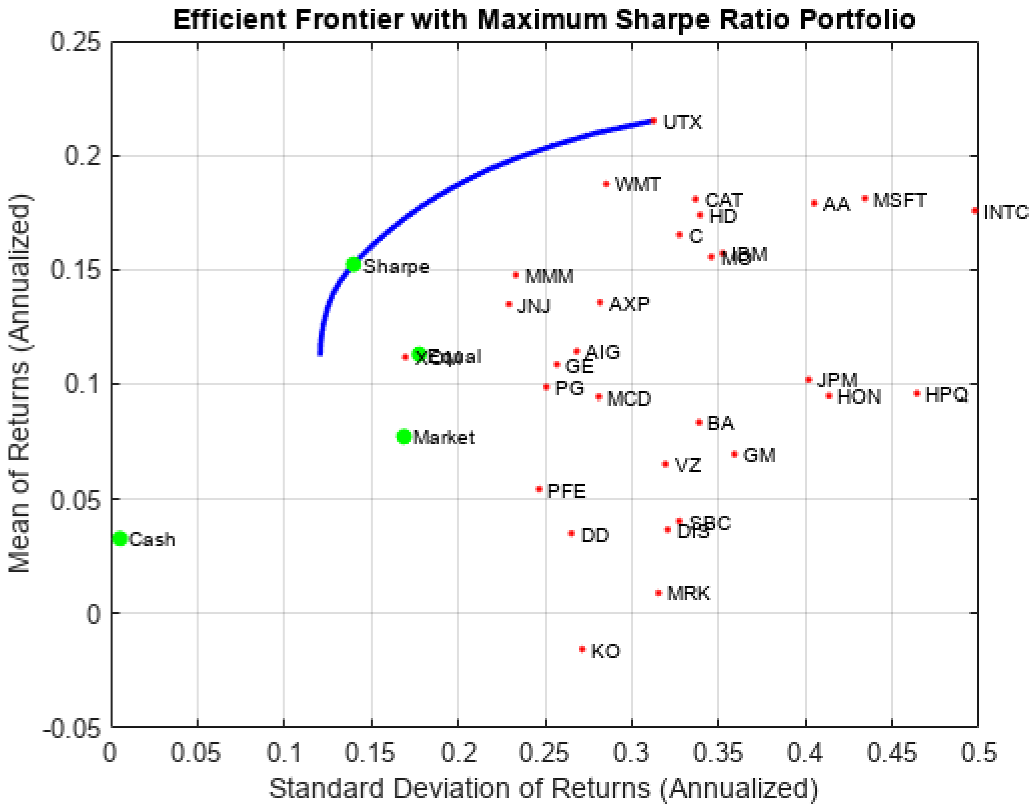
```
TrackingError = 0.05/sqrt(12);
TrackingPort = zeros(30, 1);
TrackingPort(ii) = 1;
TrackingPort = (1/sum(TrackingPort))*TrackingPort;

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 the 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 on page 4-170) 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 the 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'});
```



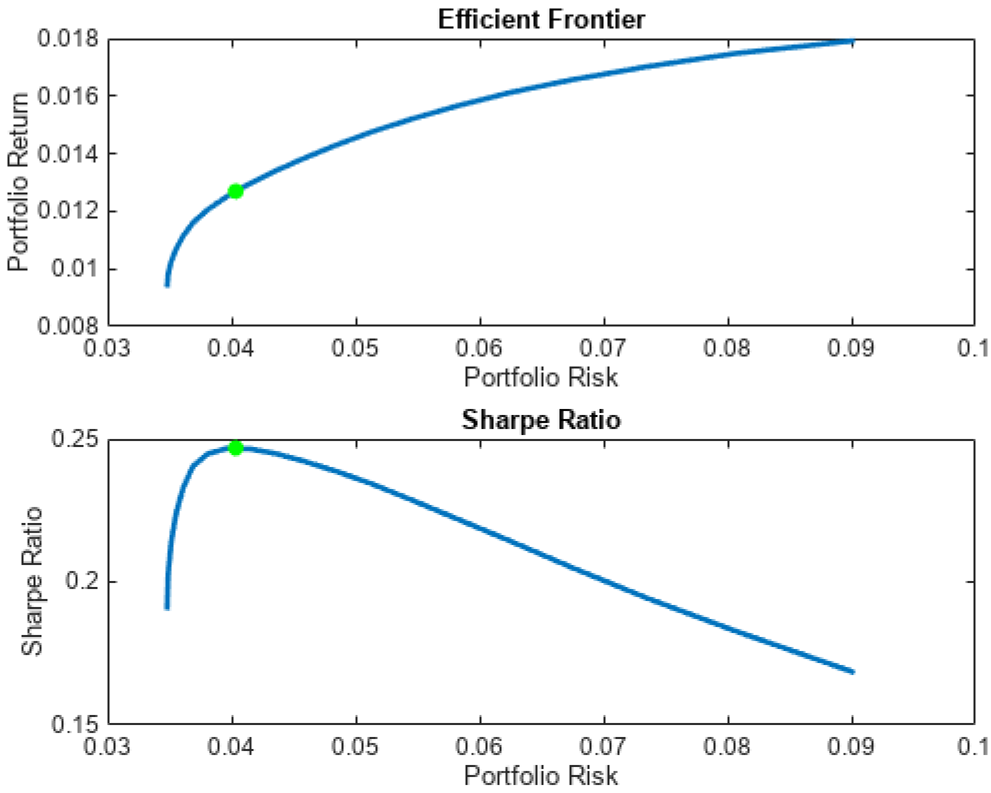
```
% Set up a dataset object that contains the portfolio that maximizes the Sharpe ratio.

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
    AA      1.9674e-15
    AIG     1.9059e-15
    AXP     1.0567e-15
    BA      5.6013e-16
    C       2.0055e-15
    CAT     2.9716e-15
    DD      3.1346e-16
    DIS     5.9556e-16
    GE      5.6145e-15
    GM      7.3246e-16
    HD      2.0342e-11
    HON      3.309e-16
    HPQ     2.0307e-15
    IBM     8.8746e-15
    INTC        2.6638
    JNJ         9.0044
    JPM     5.4026e-16
    KO      2.4602e-16
    MCD     8.7472e-16
    MMM         15.502
    MO          13.996
    MRK     3.2239e-16
    MSFT        4.4777
    PFE     7.3425e-16
    PG          7.4588
    SBC     3.8329e-16
    UTX         6.0056
    VZ      3.9079e-16
    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.

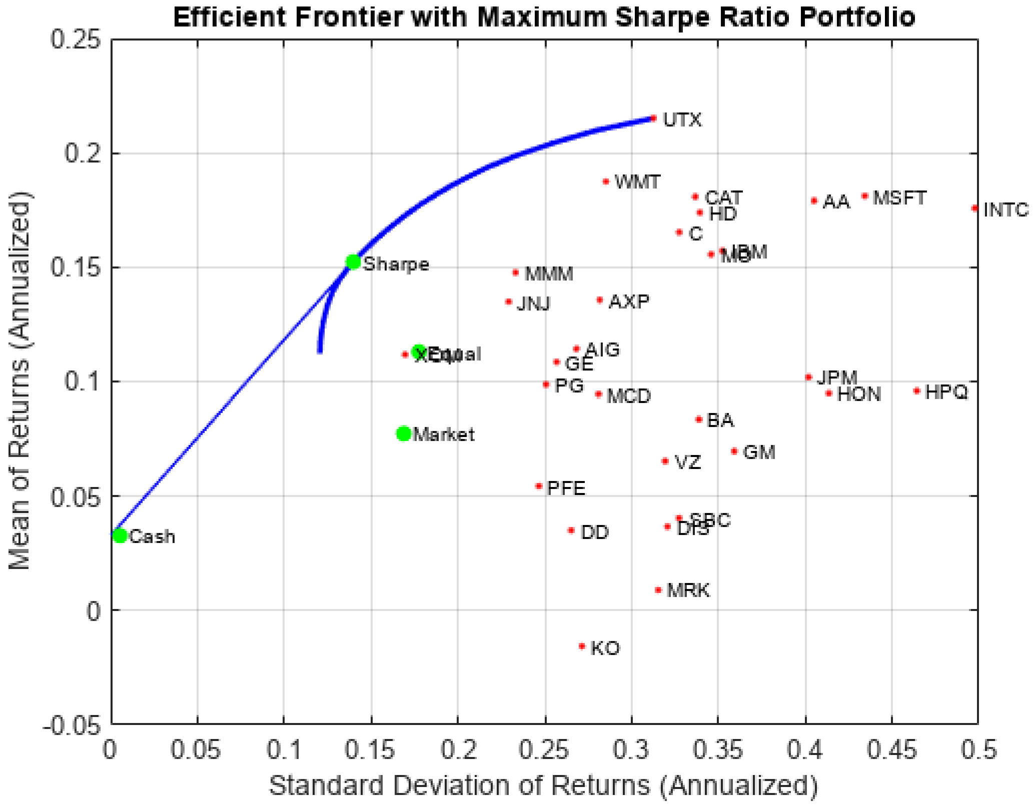
```
psratio = (pret - p.RiskFreeRate) ./ prsk;
ssratio = (sret - p.RiskFreeRate) / srsk;

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 showing that the 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'});
```



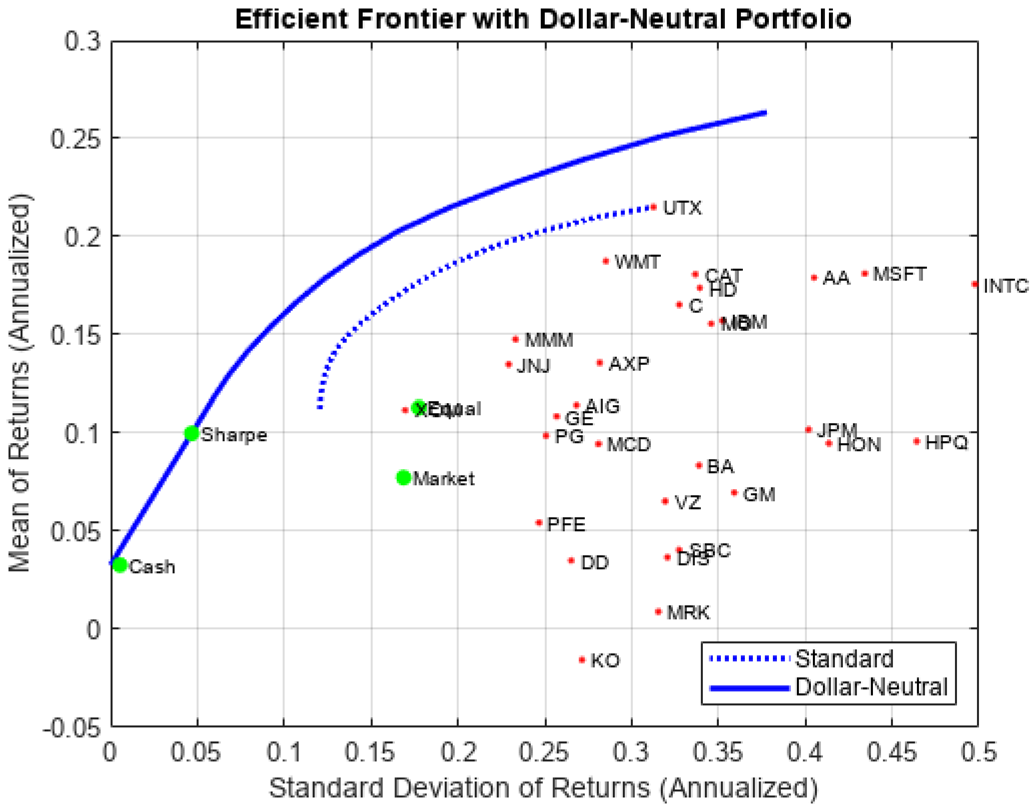
Dollar-Neutral Hedge-Fund Structure

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 the 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-Neu
```

```
Dollar-Neutral Portfolio with Maximum Sharpe Ratio
            Weight      Long        Short
    AA       0.5088      0.5088          0
    AIG      3.0394      3.0394          0
    AXP     0.92797     0.92797          0
    BA      -3.4952           0     3.4952
    C        14.003      14.003          0
    CAT      3.7261     3.7261         0
    DD      -18.063          0    18.063
    DIS     -4.8236          0    4.8236
    GE      -3.6178          0    3.6178
    GM      -3.7211          0    3.7211
    HD        1.101      1.101         0
    HON     -1.4349          0    1.4349
    HPQ     0.09909    0.09909         0
    IBM     -8.0585          0    8.0585
    INTC     1.7693     1.7693         0
    JNJ      1.3696     1.3696         0
    JPM     -2.5271          0    2.5271
    KO      -14.205          0    14.205
    MCD        3.91       3.91         0
    MMM      7.5995     7.5995         0
    MO       4.0856     4.0856         0
    MRK       3.747      3.747         0
    MSFT     4.0769     4.0769         0
    PFE      -9.096          0     9.096
    PG       1.6493     1.6493         0
    SBC     -5.2547          0    5.2547
    UTX      5.7454     5.7454         0
    VZ       -2.438          0     2.438
    WMT     0.84844    0.84844         0
    XOM      18.529     18.529         0

Confirm Dollar-Neutral Portfolio
  (Net, Long, Short)
-0.0000 76.7350 76.7350
```

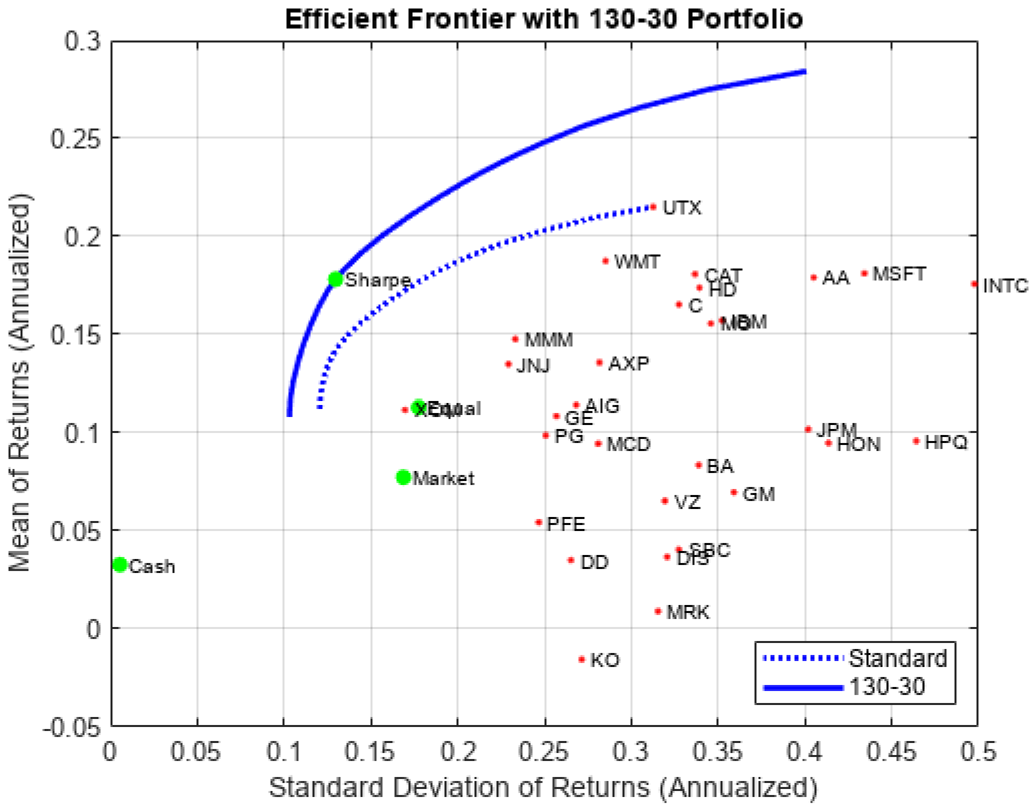
130/30 Fund Structure

Finally, the turnover constraints are 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 the 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), 100*Lev
```

```
130-30 Portfolio with Maximum Sharpe Ratio
            Weight      Long      Short
    DD      -9.5565         0     9.5565
    HON     -6.0244         0     6.0244
    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. *Active Portfolio Management.* 2nd ed.,2000.

2 H. M. Markowitz. "Portfolio Selection." *Journal of Finance.* Vol. 1, No. 1, pp. 77-91, 1952.

3 J. Lintner. "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, 1965.

4 H. M. Markowitz. *Portfolio Selection: Efficient Diversification of Investments.* John Wiley & Sons, Inc., 1959.

5 W. F. Sharpe. "Mutual Fund Performance." *Journal of Business.* Vol. 39, No. 1, Part 2, pp. 119-138, 1966.

6 J. Tobin. "Liquidity Preference as Behavior Towards Risk." *Review of Economic Studies.* Vol. 25, No.1, pp. 65-86, 1958.

7 J. L. Treynor and F. Black. "How to Use Security Analysis to Improve Portfolio Selection." *Journal of Business.* Vol. 46, No. 1, pp. 68-86, 1973.

Local 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 | setBounds | addGroups | setAssetMoments | estimateAssetMoments | estimateBounds | plotFrontier | estimateFrontierLimits | estimateFrontierByRisk | estimatePortRisk

Related Examples

- “Creating the Portfolio Object” on page 4-24
- “Working with Portfolio Constraints Using Defaults” on page 4-57
- “Validate the Portfolio Problem for Portfolio Object” on page 4-90
- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-94
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-118
- “Postprocessing Results to Set Up Tradable Portfolios” on page 4-126
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-183
- “Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215
- “Portfolio Optimization Against a Benchmark” on page 4-195
- “Asset Allocation Case Study” on page 4-172
- “Portfolio Analysis with Turnover Constraints” on page 4-204
- “Leverage in Portfolio Optimization with a Risk-Free Asset” on page 4-210
- “Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215
- “Portfolio Optimization Using Factor Models” on page 4-224
- “Portfolio Optimization Using Social Performance Measure” on page 4-257
- “Diversify Portfolios Using Custom Objective” on page 4-329

More About

- “Portfolio Object” on page 4-19
- “Portfolio Optimization Theory” on page 4-3

External Websites

- Getting Started with Portfolio Optimization (4 min 13 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 with a `Portfolio` object 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 `table` 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 = table('RowNames', Asset);
Blotter.Price = Price;
Blotter.InitHolding = Holding;
Wealth = sum(Blotter.Price .* Blotter.InitHolding);
Blotter.InitPort = (1/Wealth)*(Blotter.Price .* Blotter.InitHolding);
Blotter.UnitCost = UnitCost;
Blotter
```

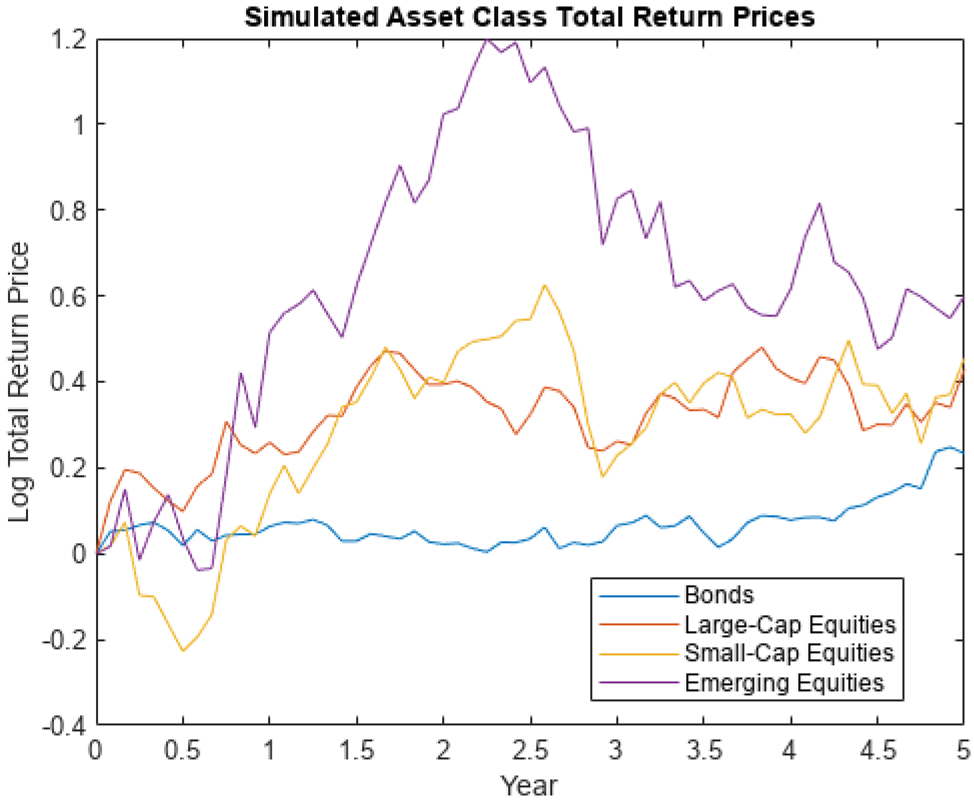
```
Blotter=4×4 table
                          Price    InitHolding    InitPort    UnitCost
                          _____    ___________    ________    ________

    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 months)
[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 object.

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: {'Bonds'  'Large-Cap Equities'  'Small-Cap Equities'  'Emerging Equities'}
         InitPort: [4x1 double]
      AInequality: []
      bInequality: []
        AEquality: []
        bEquality: []
       LowerBound: [4x1 double]
       UpperBound: []
      LowerBudget: 1
      UpperBudget: 1
      GroupMatrix: [2x4 double]
       LowerGroup: []
       UpperGroup: [2x1 double]
           GroupA: []
           GroupB: []
       LowerRatio: []
       UpperRatio: []
     MinNumAssets: []
     MaxNumAssets: []
        BoundType: [4x1 categorical]
```

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.

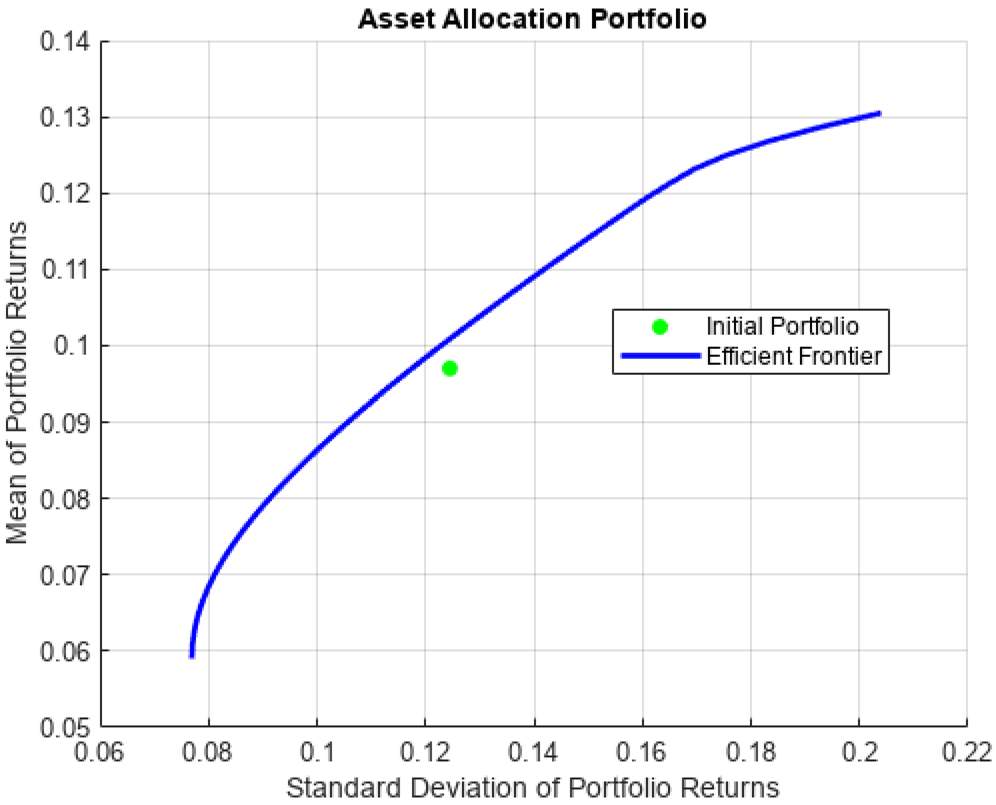
```
[lb, ub] = estimateBounds(p);
display([lb, ub])
```

```
    0.1500    1.0000
         0    0.8500
         0    0.8500
         0    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: {'Bonds'  'Large-Cap Equities'  'Small-Cap Equities'  'Emerging Equities'}
         InitPort: [4x1 double]
      AInequality: []
      bInequality: []
```

```
        AEquality: []
        bEquality: []
       LowerBound: [4x1 double]
       UpperBound: []
      LowerBudget: 1
      UpperBudget: 1
      GroupMatrix: [2x4 double]
       LowerGroup: []
       UpperGroup: [2x1 double]
           GroupA: []
           GroupB: []
       LowerRatio: []
       UpperRatio: []
     MinNumAssets: []
     MaxNumAssets: []
        BoundType: [4x1 categorical]
```

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 show 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 = 4×1

    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 = 4×3

    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 using `setTurnover`:

```
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=4×7 table
                          Price    InitHolding    InitPort    UnitCost     Port      Buy       Se
                          _____    ___________    ________    ________    _______    ____    ____

    Bonds                  52.4       42938         0.3        0.001      0.18787       0     0.1
    Large-Cap Equities    122.7       24449         0.4        0.001          0.4       0
    Small-Cap Equities     35.2       42612         0.2        0.001      0.16213       0    0.03
    Emerging Equities      46.9       15991         0.1        0.004         0.25    0.15
```

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 is 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 the 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 *ad 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.

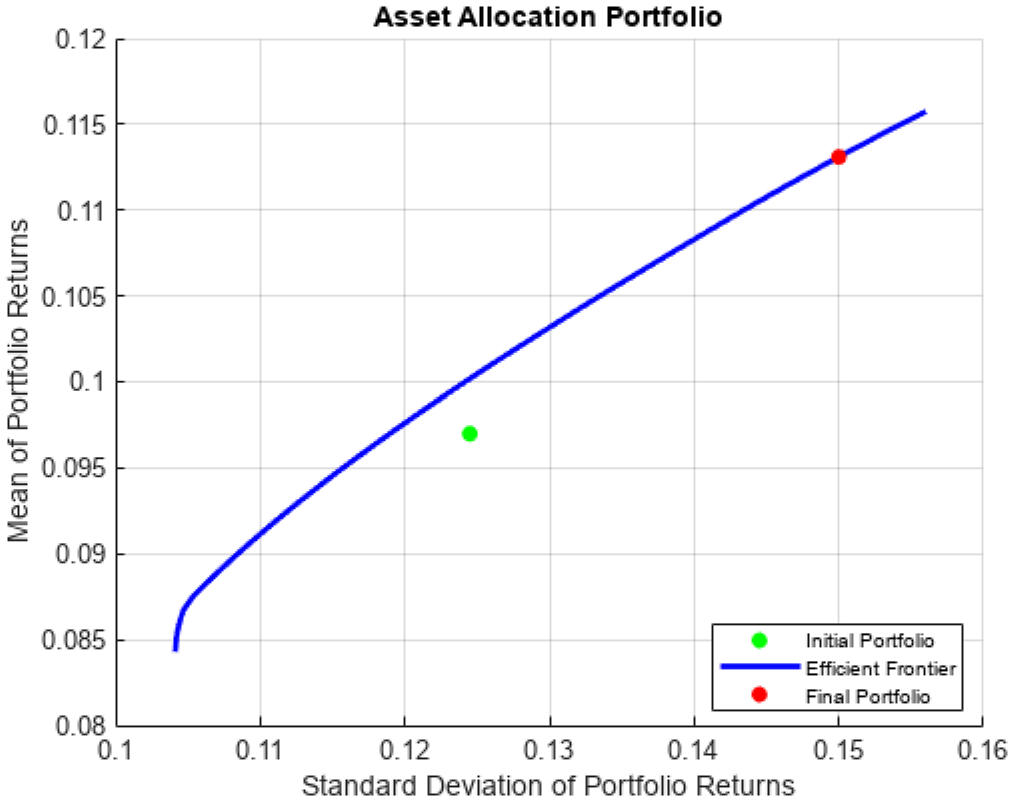
The final result is a blotter that contains proposed trades to get from your current portfolio to a moderate-risk portfolio. To make the trade, you would need to sell 16,049 shares of your bond asset and 8,069 shares of your small-cap equity asset and would need to purchase 23,986 shares of your emerging equities asset.

```
display(Blotter)
```

```
Blotter=4×7 table
                          Price    InitHolding    InitPort     Port      Holding    BuyShare    S
                          _____    ___________    ________    _______    _______    ________    _

    Bonds                  52.4       42938          0.3      0.18787     26889            0
    Large-Cap Equities    122.7       24449          0.4          0.4     24449            0
    Small-Cap Equities     35.2       42612          0.2      0.16213     34543            0
    Emerging Equities      46.9       15991          0.1         0.25     39977        23986
```

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', 'best');
set(h, 'Fontsize', 8);
hold off
```



Local 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(1));
fprintf('Maximum Efficient Portfolio Return %6.2f %%  %6.2f %%\n',100*pret(2),100*qret(2));
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 | setBounds | addGroups | setAssetMoments | estimateAssetMoments | estimateBounds | plotFrontier | estimateFrontierLimits | estimateFrontierByRisk | estimatePortRisk

Related Examples

- “Creating the Portfolio Object” on page 4-24
- “Working with Portfolio Constraints Using Defaults” on page 4-57

- “Validate the Portfolio Problem for Portfolio Object” on page 4-90
- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-94
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-118
- “Postprocessing Results to Set Up Tradable Portfolios” on page 4-126
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-183
- “Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215
- “Portfolio Optimization Against a Benchmark” on page 4-195
- “Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
- “Portfolio Analysis with Turnover Constraints” on page 4-204
- “Leverage in Portfolio Optimization with a Risk-Free Asset” on page 4-210
- “Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215
- “Portfolio Optimization Using Factor Models” on page 4-224
- “Portfolio Optimization Using Social Performance Measure” on page 4-257
- “Diversify Portfolios Using Custom Objective” on page 4-329

More About

- “Portfolio Object” on page 4-19
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-17

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Portfolio Optimization with Semicontinuous and Cardinality Constraints

This example shows how to use a Portfolio object to directly handle semicontinuous and cardinality constraints when performing portfolio optimization. Portfolio optimization finds the asset allocation that maximizes the return or minimizes the risk, subject to a set of investment constraints. The Portfolio class in Financial Toolbox™ is designed and implemented based on the Markowitz Mean-Variance Optimization framework. The Mean-Variance Optimization framework handles problems where the return is the expected portfolio return, and the risk is the variance of portfolio returns. Using the Portfolio class, you can minimize the risk on the efficient frontier (EF), maximize the return on the EF, maximize the return for a given risk, and minimize the risk for a given return. You can also use PortfolioCVaR or PortfolioMAD classes in Financial Toolbox™ to specify semicontinuous and cardinality constraints. Such optimization problems integrate with constraints such as group, linear inequality, turnover, and tracking error constraints. These constraints are formulated as nonlinear programming (NLP) problems with continuous variables represented as the asset weights x_i.

Semicontinuous and cardinality constraints are two other common categories of portfolio constraints that are formulated mathematically by adding the binary variables v_i.

- A *semicontinuous constraint* confines the allocation of an asset. For example, you can use this constraint to confine the allocated weight of an allocated asset to between 5% and 50%. By using this constraint, you can avoid very small or large positions to minimize the churns and operational costs. To mathematically formulate this type of constraint, a binary variable v_i is needed, where v_i is 0 or 1. The value 0 indicates that asset *i* is not allocated and the value 1 indicates that asset *i* is allocated. The mathematical form is $\mathrm{lb} * v_i \le x_i \le \mathrm{ub} * v_i$, where v_i is 0 or 1. Specify this type of constraint as a 'Conditional' BoundType in the Portfolio class using the setBounds function.
- A *cardinality constraint* limits the number of assets in the optimal allocation, For example, for a portfolio with a universe of 100 assets, you can specify an optimal portfolio allocation between 20 and 40 assets. This capability helps limit the number of positions, and thus reduce operational costs. To mathematically formulate this type of constraint, binary variables represented as v_i are needed, where v_i is 0 or 1. The value 0 indicates that asset *i* is not allocated and the value 1 indicates that asset *i* is allocated. The mathematical form is $\mathrm{MinNumAssets} \le \sum_1^{\mathrm{NumAssets}} v_i \le \mathrm{MaxNumAssets}$, where v_i is 0 or 1. Specify this type of constraint by setting the 'MinNumAssets' and 'MaxNumAssets'constraints in the Portfolio class using the setMinMaxNumAssets function.

For more information on semicontinuous and cardinality constraints, see “Algorithms” on page 15-1399.

When semicontinuous and cardinality constraints are used for portfolio optimization, this leads to mixed integer nonlinear programming problems (MINLP). The Portfolio class allows you to configure these two constraints, specifically, semicontinuous constraints using setBounds with 'Conditional' BoundType, and cardinality constraints using setMinMaxNumAssets. The Portfolio class automatically formulates the mathematical problems and validates the specified constraints. The Portfolio class also provides built-in MINLP solvers and flexible solver options for you to tune the solver performance using the setSolverMINLP function.

This example demonstrates a `Portfolio` object with semicontinuous and cardinality constraints and uses the `BlueChipStockMoments` dataset, which has a universe of 30 assets.

```
load BlueChipStockMoments
numAssets = numel(AssetList)
```

```
numAssets = 30
```

Limit the Minimum Weight for Each Allocated Asset

Create a fully invested portfolio with only long positions: $x_i \geq 0$ and $\text{sum}(x_i) = 1$. These are configured with `setDefaultConstraints`.

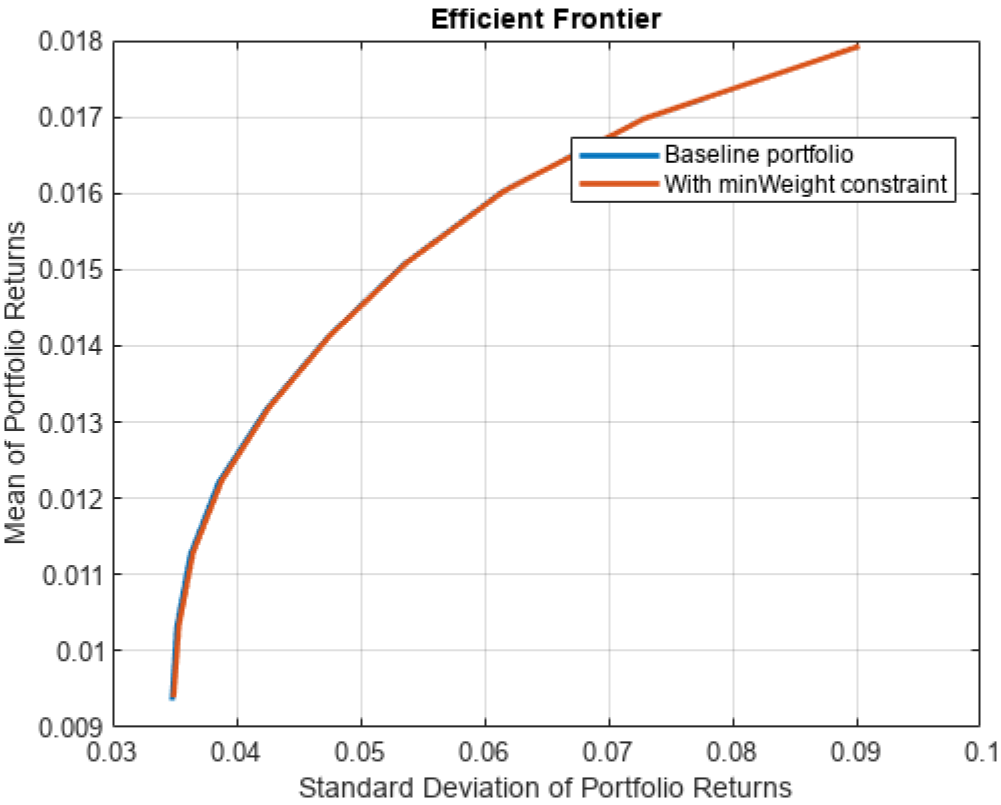
```
p = Portfolio('AssetList', AssetList,'AssetCovar', AssetCovar, 'AssetMean', AssetMean);
p = setDefaultConstraints(p);
```

Suppose that you want to avoid very small positions to minimize the churn and operational costs. Add another constraint to confine the allocated positions to be no less than 5%, by setting the constraints $x_i = 0$ or $x_i \geq 0.05$ using `setBounds` with a `'Conditional'` `BoundType`.

```
pWithMinWeight = setBounds(p, 0.05, 'BoundType', 'Conditional');
```

Plot the efficient frontiers for both `Portfolio` objects.

```
wgt = estimateFrontier(p);
wgtWithMinWeight = estimateFrontier(pWithMinWeight);
figure(1);
plotFrontier(p, wgt); hold on;
plotFrontier(pWithMinWeight, wgtWithMinWeight); hold off;
legend('Baseline portfolio', 'With minWeight constraint', 'location', 'best');
```



The figure shows that the two `Portfolio` objects have almost identical efficient frontiers. However, the one with the minimum weight requirement is more practical, since it prevents the close-to-zero positions.

Check the optimal weights for the portfolio with default constraints to see how many assets are below the 5% limit for each optimal allocation.

```
toler = eps;
sum(wgt>toler & wgt<0.05)
```

```
ans = 1×10

     5     7     5     4     2     3     4     2     0     0
```

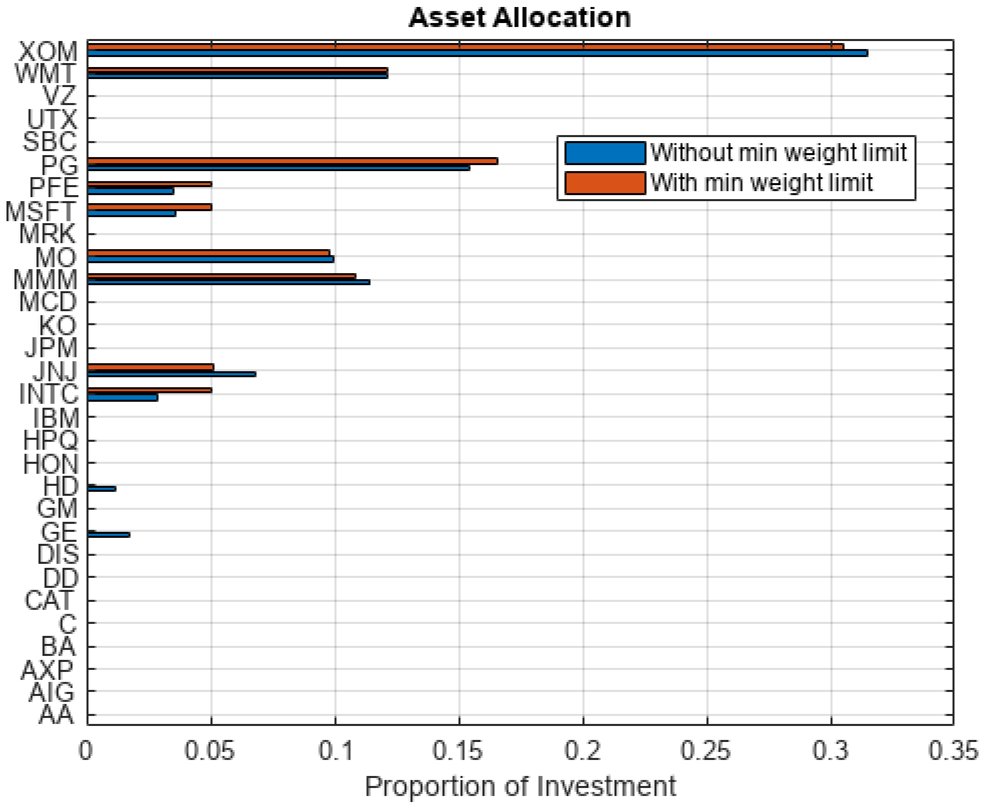
Use `estimateFrontierByReturn` to investigate the portfolio compositions for a target return on the frontier for both cases.

```
targetRetn = 0.011;
pwgt = estimateFrontierByReturn(p, targetRetn);
pwgtWithMinWeight = estimateFrontierByReturn(pWithMinWeight, targetRetn);
```

Plot the composition of the two `Portfolio` objects for the universe of 30 assets.

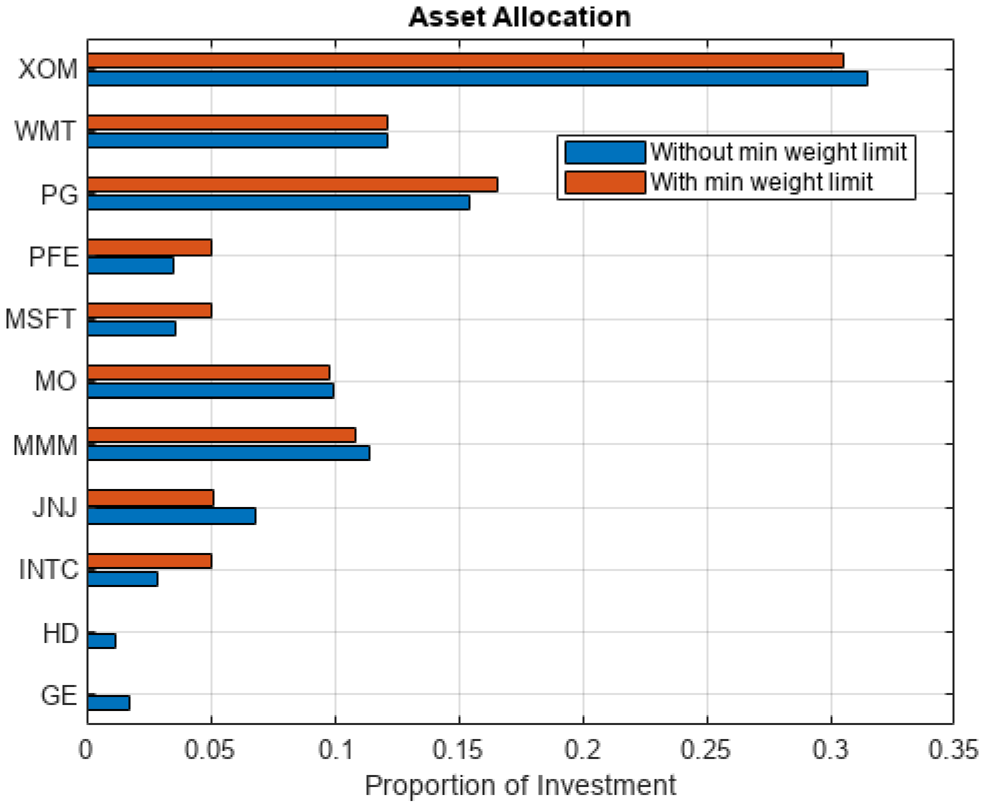
```
figure(2);
barh([pwgt, pwgtWithMinWeight]);
grid on
xlabel('Proportion of Investment')
```

```
yticks(1:p.NumAssets);
yticklabels(p.AssetList);
title('Asset Allocation');
legend('Without min weight limit', 'With min weight limit', 'location', 'best');
```



Show only the allocated assets.

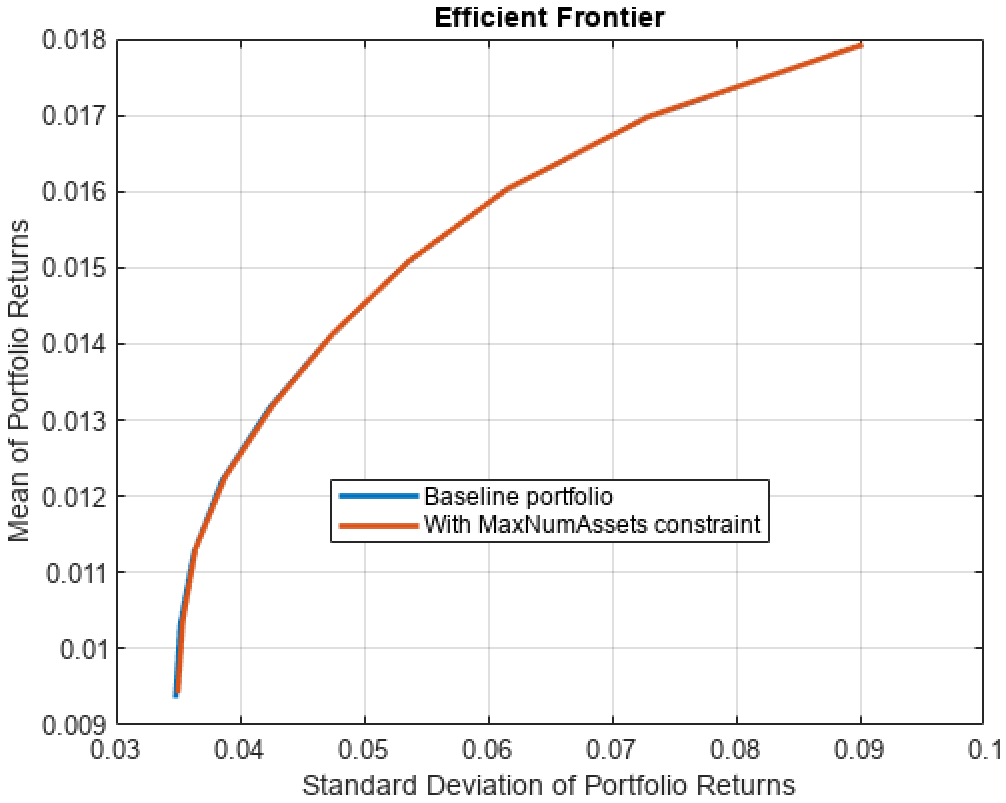
```
idx = (pwgt>toler) | (pwgtWithMinWeight>toler);

barh([pwgt(idx), pwgtWithMinWeight(idx)]);
grid on
xlabel('Proportion of Investment')
yticks(1:sum(idx));
yticklabels(p.AssetList(idx));
title('Asset Allocation');
legend('Without min weight limit', 'With min weight limit', 'location', 'best');
```



Limit the Maximum Number of Assets to Allocate

Use `setMinMaxNumAssets` to set the maximum number of allocated assets for the `Portfolio` object. Suppose that you want no more than eight assets invested in the optimal portfolio. To do this with a `Portfolio` object, use `setMinMaxNumAssets`.

```
pWithMaxNumAssets = setMinMaxNumAssets(p, [], 8);

wgt = estimateFrontier(p);
wgtWithMaxNumAssets = estimateFrontier(pWithMaxNumAssets);
plotFrontier(p, wgt); hold on;
plotFrontier(pWithMaxNumAssets, wgtWithMaxNumAssets); hold off;
legend('Baseline portfolio', 'With MaxNumAssets constraint', 'location', 'best');
```

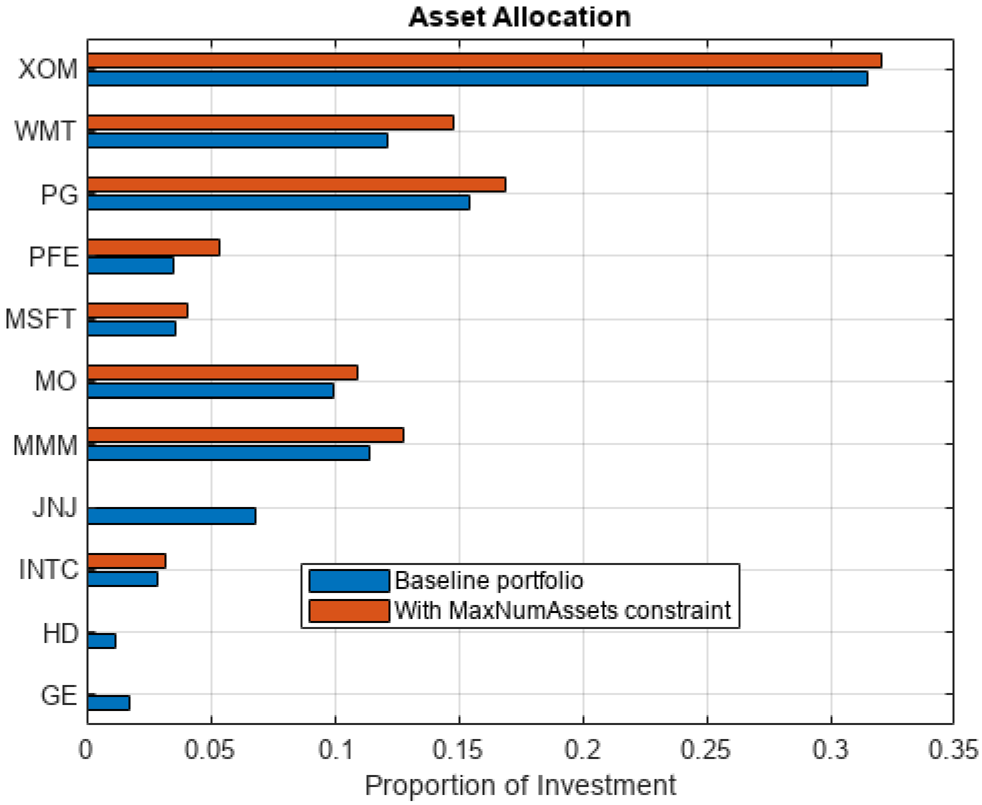


Use `estimateFrontierByReturn` to find the allocation that minimizes the risk on the frontier for the given target return.

```
pwgtWithMaxNum = estimateFrontierByReturn(pWithMaxNumAssets, targetRetn);
```

Plot the composition of the two `Portfolio` objects for the universe of 30 assets.

```
idx = (pwgt>toler) | (pwgtWithMaxNum>toler);

barh([pwgt(idx), pwgtWithMaxNum(idx)]);
grid on
xlabel('Proportion of Investment')
yticks(1:sum(idx));
yticklabels(p.AssetList(idx));
title('Asset Allocation');
legend('Baseline portfolio', 'With MaxNumAssets constraint', 'location', 'best');
```



```
sum(abs(pwgt)>toler)
```

```
ans = 11
```

Count the total number of allocated assets to verify that only eight assets at most are allocated.

```
sum(abs(pwgtWithMaxNum)>toler)
```

```
ans = 8
```

Limit the Minimum and Maximum Number of Assets to Allocate

Suppose that you want to set both the lower and upper bounds for the number of assets to allocate in a portfolio, given the universe of assets. Use `setBounds` to specify the allowed number of assets to allocate as from 5 through 10, and the allocated weight as no less than 5%.

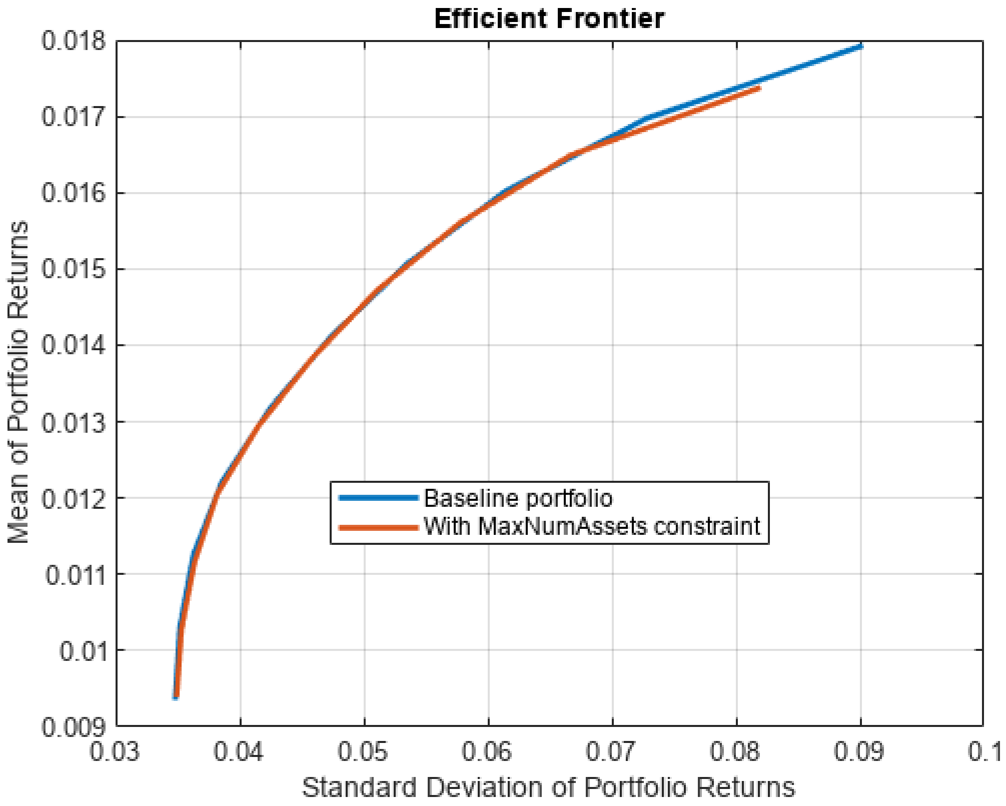
```
p1 = setMinMaxNumAssets(p, 5, 10);
p1 = setBounds(p1, 0.05, 'BoundType', 'conditional');
```

If an asset is allocated, it is necessary to clearly define the minimum weight requirement for that asset. This is done using `setBounds` with a `'Conditional'` `BoundType`. Otherwise, the optimizer cannot evaluate which assets are allocated and cannot formulate the `MinNumAssets` constraint. For more details, see “Conditional Bounds with LowerBound Defined as Empty or Zero” on page 4-140.

Plot the efficient frontier to compare this portfolio to the baseline portfolio, which has only default constraints.

```
wgt = estimateFrontier(p);
wgt1 = estimateFrontier(p1);
```

```
plotFrontier(p, wgt); hold on;
plotFrontier(p1, wgt1); hold off;
legend('Baseline portfolio', 'With MaxNumAssets constraint', 'location', 'best');
```



Asset Allocation for an Equal-Weighted Portfolio

Create an equal-weighted portfolio using both `setBounds` and `setMinMaxNumAssets` functions.

```
numAssetsAllocated = 8;
weight= 1/numAssetsAllocated;
p2 = setBounds(p, weight, weight, 'BoundType', 'conditional');
p2 = setMinMaxNumAssets(p2, numAssetsAllocated, numAssetsAllocated);
```

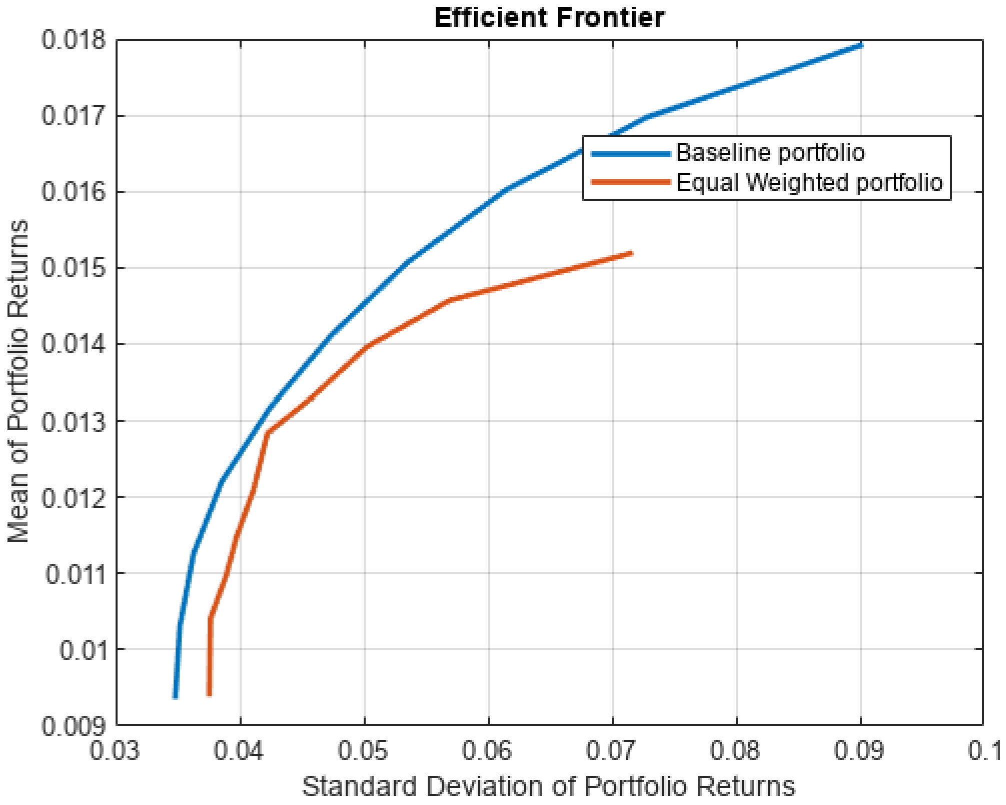
When any one, or any combination of `'Conditional'` `BoundType`, `MinNumAssets`, or `MaxNumAssets` are active, the optimization problem is formulated as a mixed integer nonlinear programming (MINLP) problem. The `Portfolio` class automatically constructs the MINLP problem based on the specified constraints.

When working with a `Portfolio` object, you can select one of three solvers using the `setSolverMINLP` function. In this example, instead of using default MINLP solver options, customize the solver options to help with a convergence issue. Use a large number (`50`) for `'MaxIterationsInactiveCut'` with `setSolverMINLP`, instead of the default value of `30` for '`MaxIterationsInactiveCut'`. The value `50` works well in finding the efficient frontier of optimal asset allocation.

```
p2 = setSolverMINLP(p2, 'OuterApproximation', 'MaxIterationsInactiveCut', 50);
```

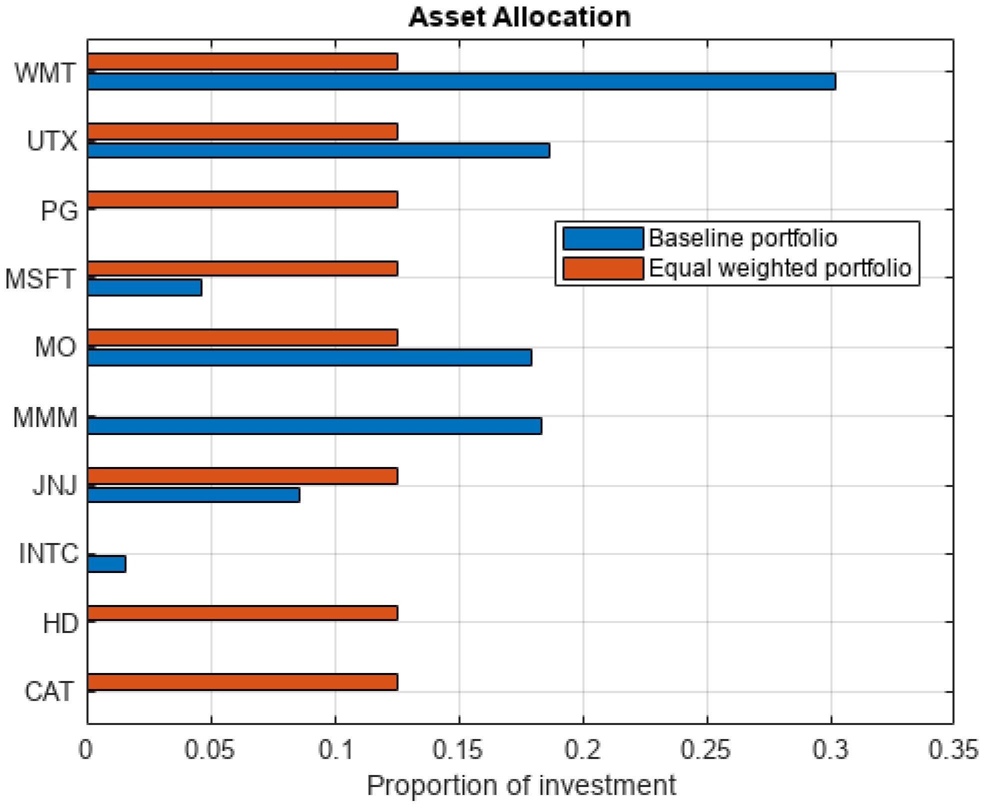
Plot the efficient frontiers for the baseline and equal-weighted portfolios.

```
wgt = estimateFrontier(p);
wgt2 = estimateFrontier(p2);
plotFrontier(p, wgt); hold on;
plotFrontier(p2, wgt2); hold off;
legend('Baseline portfolio', 'Equal Weighted portfolio', 'location', 'best');
```



Use `estimateFrontierByRisk` to optimize for a specific risk level, in this case `.05`, to determine what allocation maximizes the portfolio return.

```
targetRisk = 0.05;
pwgt = estimateFrontierByRisk(p, targetRisk);
pwgt2 = estimateFrontierByRisk(p2, targetRisk);

idx = (pwgt>toler) | (pwgt2>toler);
barh([pwgt(idx), pwgt2(idx)]);
grid on
xlabel('Proportion of investment')
yticks(1:sum(idx));
yticklabels(p.AssetList(idx));
title('Asset Allocation');
legend('Baseline portfolio', 'Equal weighted portfolio', 'location', 'best');
```



Use 'Conditional' BoundType, MinNumAssets, and MaxNumAssets Constraints with Other Constraints

You can define other constraints for a Portfolio object using the set functions. These other constraints for a Portfolio object, such as group, linear inequality, turnover, and tracking error can be used together with the 'Conditional' BoundType, 'MinNumAssets', and 'MaxNumAssets' constraints. For example, specify a tracking error constraint using setTrackingError.

```
ii = [15, 16, 20, 21, 23, 25, 27, 29, 30];    % indexes of assets to include in tracking portfoli
trackingPort(ii) = 1/numel(ii);
q = setTrackingError(p, 0.5, trackingPort);
```

Then use setMinMaxNumAssets to add a constraint to limit maximum number of assets to invest.

```
q = setMinMaxNumAssets(q, [], 8);
```

On top of these previously specified constraints, use setBounds to add a constraint to limit the weight for the allocated assets. You can use constraints with mixed BoundType values, where 'Simple' means lb ≤ x_i ≤ ub and 'Conditional' means x_i = 0 or lb ≤ x_i ≤ ub.

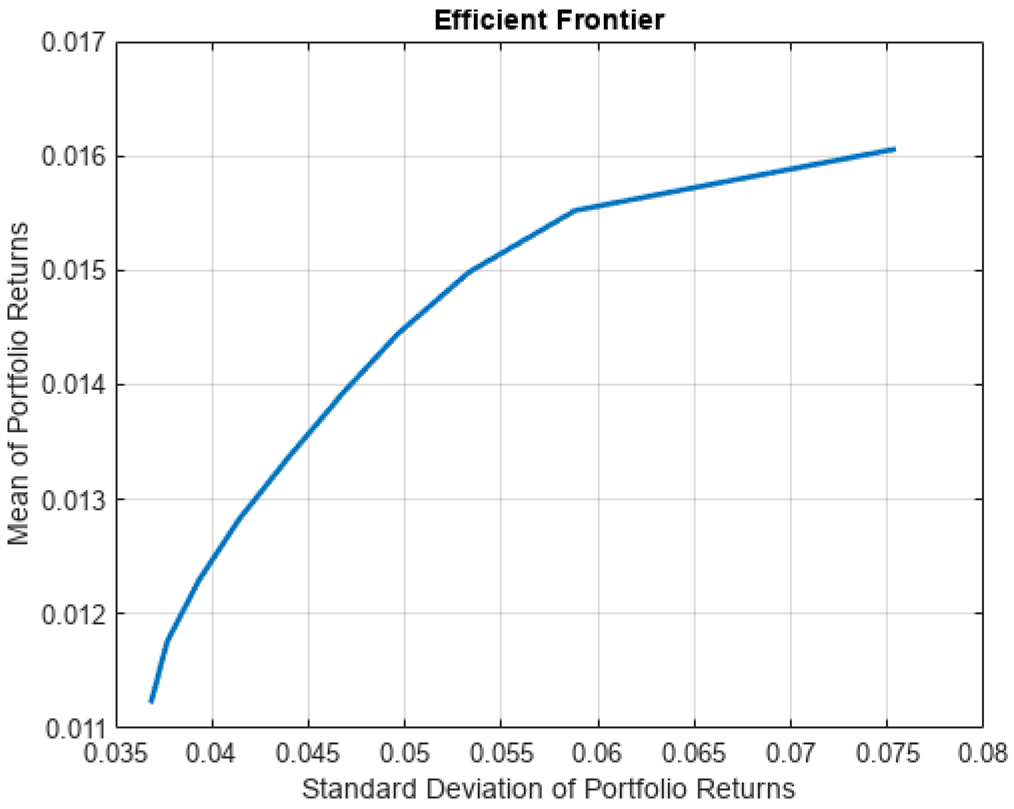
Allow the assets in trackingPort to have the BoundType value 'Conditional' in the optimum allocation.

```
lb = zeros(q.NumAssets, 1);
ub = zeros(q.NumAssets, 1)*0.5;
lb(ii) = 0.1;
ub(ii) = 0.3;
```

```
boundType = repmat("simple",q.NumAssets,1);
boundType(ii) = "conditional";
q = setBounds(q, lb, ub, 'BoundType',boundType);
```

Plot the efficient frontier:

```
plotFrontier(q);
```



Use `estimateFrontierByReturn` to find the allocation that minimizes the risk for a given return at `0.125`.

```
targetRetn = 0.0125;
pwgt = estimateFrontierByReturn(q, targetRetn);
```

Show the allocation of assets by weight.

```
idx = abs(pwgt)>eps;
assetnames = q.AssetList';
Asset = assetnames(idx);
Weight = pwgt(idx);
resultAlloc = table(Asset, Weight)
```

```
resultAlloc=7×2 table
      Asset       Weight
    _________    _______

    {'JNJ' }         0.1
    {'MMM' }     0.19503
    {'MO'  }     0.1485
    {'MSFT'}        0.1
    {'PG'  }        0.1
    {'WMT' }     0.2212
    {'XOM' }    0.13527
```

See Also

`Portfolio` | `setBounds` | `setMinMaxNumAssets` | `setSolverMINLP`

Related Examples

- “Creating the Portfolio Object” on page 4-24
- “Working with Portfolio Constraints Using Defaults” on page 4-57
- “Asset Allocation Case Study” on page 4-172
- “Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
- “Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215
- “Portfolio Optimization Using Factor Models” on page 4-224
- “Portfolio Optimization Using Social Performance Measure” on page 4-257
- “Diversify Portfolios Using Custom Objective” on page 4-329

More About

- “Portfolio Object” on page 4-19
- “Portfolio Optimization Theory” on page 4-3
- “Working with 'Conditional' BoundType, MinNumAssets, and MaxNumAssets Constraints Using Portfolio Objects” on page 4-78
- “Portfolio Object Workflow” on page 4-17

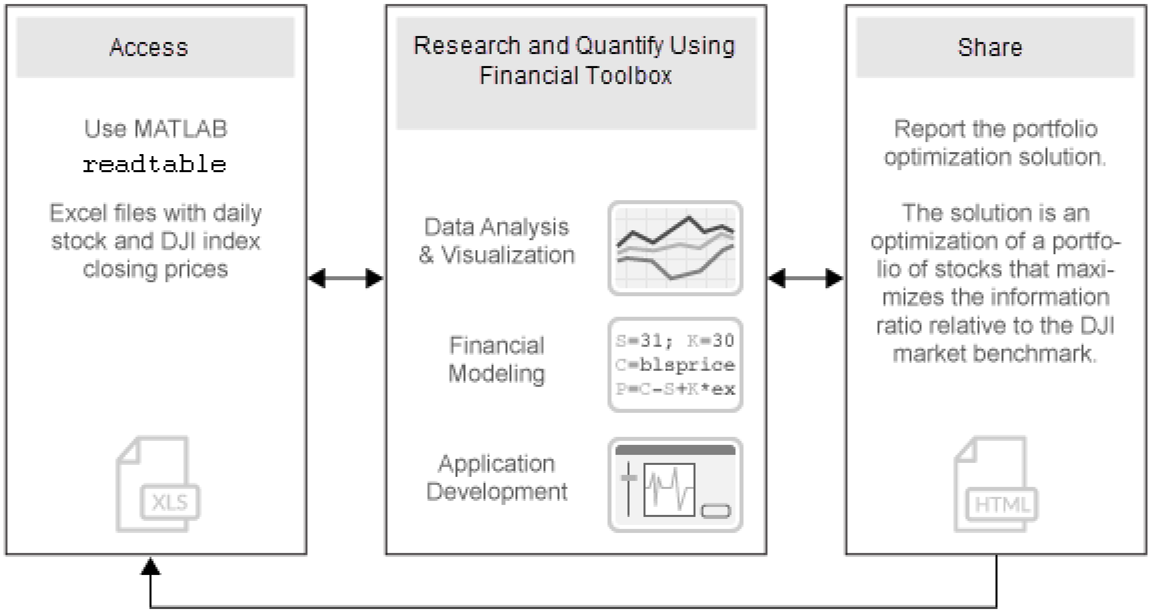
External Websites

- Getting Started with Portfolio Optimization (4 min 13 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™.

This example, in particular, demonstrates optimizing a portfolio to maximize the information ratio relative to a market benchmark. Specifically, financial data contained in a `table` is read into MATLAB® and visualizations (at both daily and annual levels) are performed. A `Portfolio` object is created with the market data using an active daily return for each asset. Using functions supporting a `Portfolio` object, the efficient frontier is calculated directly and a customized optimization problem is solved to find the asset allocation with the maximized information ratio.


Access
Use MATLAB
readtable
Excel files with daily stock and DJI index closing prices
XLS
Research and Quantify Using Financial Toolbox
Data Analysis & Visualization
Financial Modeling
S=31; K=30
C=blsprice
P=C-S+K*ex
Application Development
Share
Report the portfolio optimization solution.
HTML

The solution is an optimization of a portfolio of stocks that maximizes the information ratio relative to the DJI market benchmark.

Import Historical Data Using MATLAB®

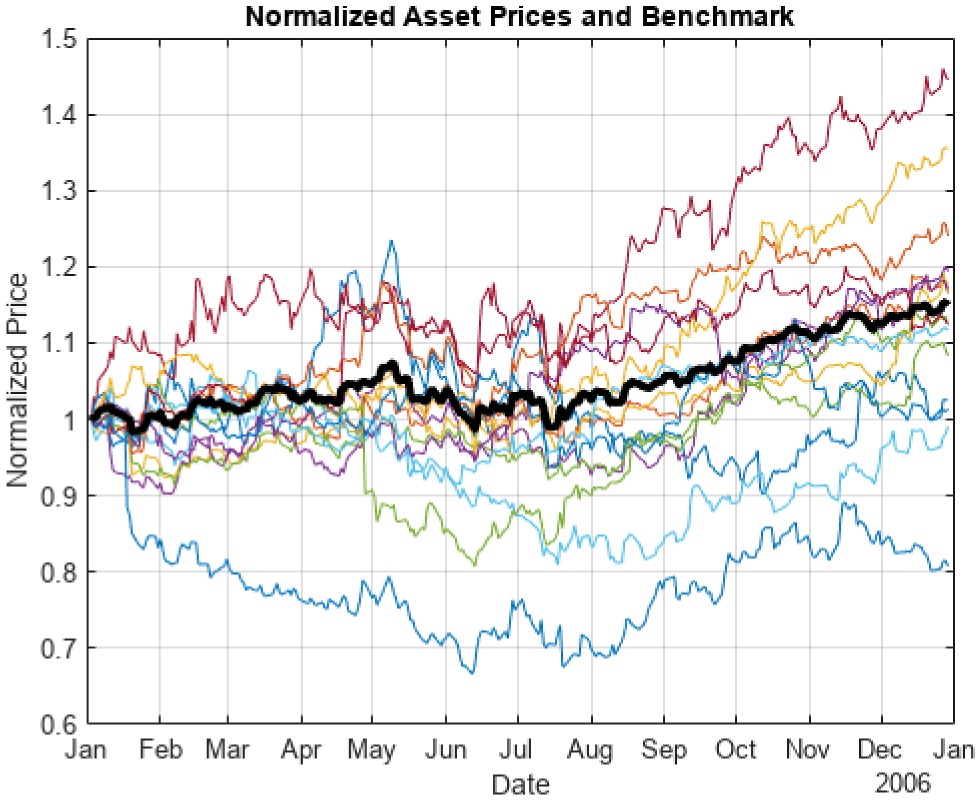
Import historical prices for the asset universe and the Dow Jones Industrial Average (DJI) market benchmark. The data is imported into a `table` from a Microsoft® Excel® spreadsheet using the MATLAB® `readtable` function.

```
data = readtable('dowPortfolio.xlsx');
head(data, 10)
```

```
       Dates        DJI      AA       AIG      AXP      BA       C        CAT      DD       DIS
    ___________    _____    _____    _____    _____    _____    _____    _____    _____    _____

    03-Jan-2006    10847    28.72    68.41    51.53    68.63    45.26    55.86    40.68    24.18
    04-Jan-2006    10880    28.89    68.51    51.03    69.34    44.42    57.29    40.46    23.77
    05-Jan-2006    10882    29.12     68.6    51.57    68.53    44.65    57.29    40.38    24.19
    06-Jan-2006    10959    29.02    68.89    51.75    67.57    44.65    58.43    40.55    24.52
    09-Jan-2006    11012    29.37    68.57    53.04    67.01    44.43    59.49    40.32    24.78
    10-Jan-2006    11012    28.44    69.18    52.88    67.33    44.57    59.25     40.2    25.09
    11-Jan-2006    11043    28.05     69.6    52.59     68.3    44.98    59.28    38.87    25.33
    12-Jan-2006    10962    27.68    69.04     52.6     67.9    45.02    60.13    38.02    25.41
    13-Jan-2006    10960     27.81     68.84      52.5      67.7     44.92     60.24     37.86     25.47
    17-Jan-2006    10896     27.97     67.84     52.03     66.93     44.47     60.85     37.75     25.15
```

Separate the asset names, asset prices, and DJI benchmark prices from the table. The visualization shows the evolution of all the asset prices normalized to start at unity, that is accumulative returns.

```
benchPrice = data.DJI;
assetNames = data.Properties.VariableNames(3:2:end); % using half of the assets for display
assetPrice = data(:,assetNames).Variables;

assetP = assetPrice./assetPrice(1, :);
benchmarkP = benchPrice / benchPrice(1);

figure;
plot(data.Dates,assetP);
hold on;
plot(data.Dates,benchmarkP,'LineWidth',3,'Color','k');
hold off;
xlabel('Date');
ylabel('Normalized Price');
title('Normalized Asset Prices and Benchmark');
grid on;
```



The bold line indicates the DJIA market benchmark.

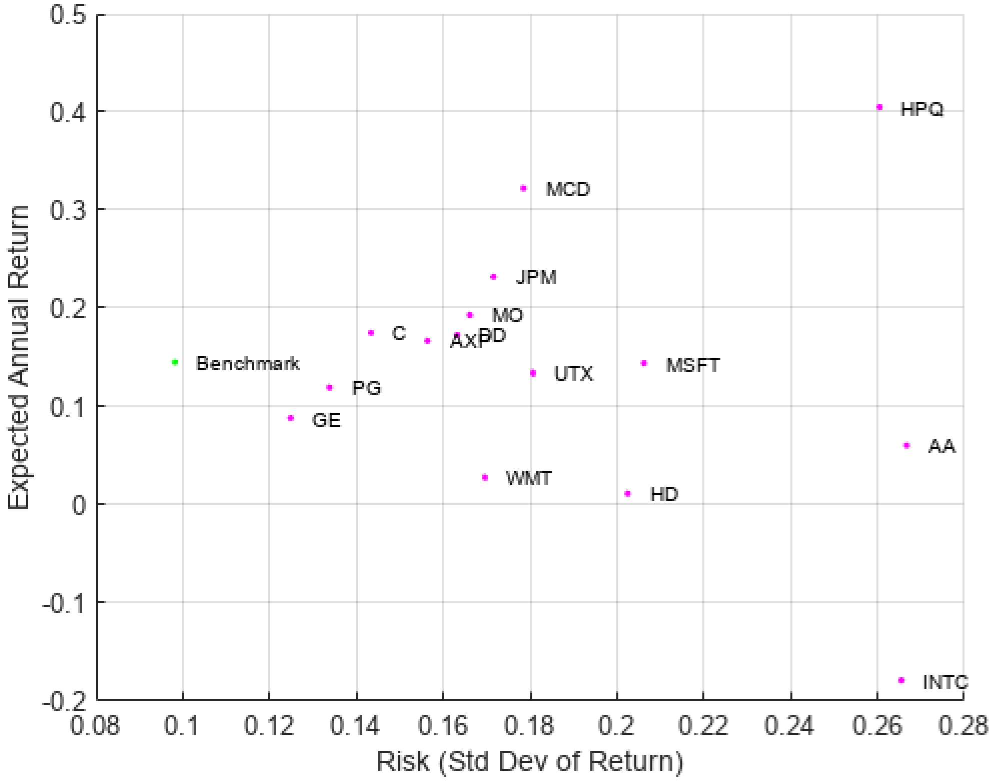
Compute Returns and Risk-Adjusted Returns

Calculate the return series from the price series and compute the 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);

benchRetn = mean(benchReturn);
benchRisk =  std(benchReturn);
assetRetn = mean(assetReturn);
assetRisk =  std(assetReturn);
```

Calculate historical statistics and plot the annual risk-return. Note that the plot is at the annual level, therefore scaling is performed on the daily returns.

```
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 Annual Return');
grid on;
```



Set Up a Portfolio Optimization

Set up a portfolio optimization problem by populating the object using `Portfolio`. Because the goal is to optimize portfolio allocation against a benchmark, the active return of each asset is computed and used in the `Portfolio` 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);
```

Set up default portfolio constraints (all weights sum to 1, no shorting, and 100% investment in risky assets).

```
p = setDefaultConstraints(p);
```

Add asset returns and covariance to the `Portfolio` object.

```
activReturn = assetReturn - benchReturn;
pAct = estimateAssetMoments(p,activReturn,'missingdata',false)
```

```
pAct = 
  Portfolio with properties:

          BuyCost: []
         SellCost: []
     RiskFreeRate: []
        AssetMean: [15x1 double]
       AssetCovar: [15x15 double]
    TrackingError: []
```

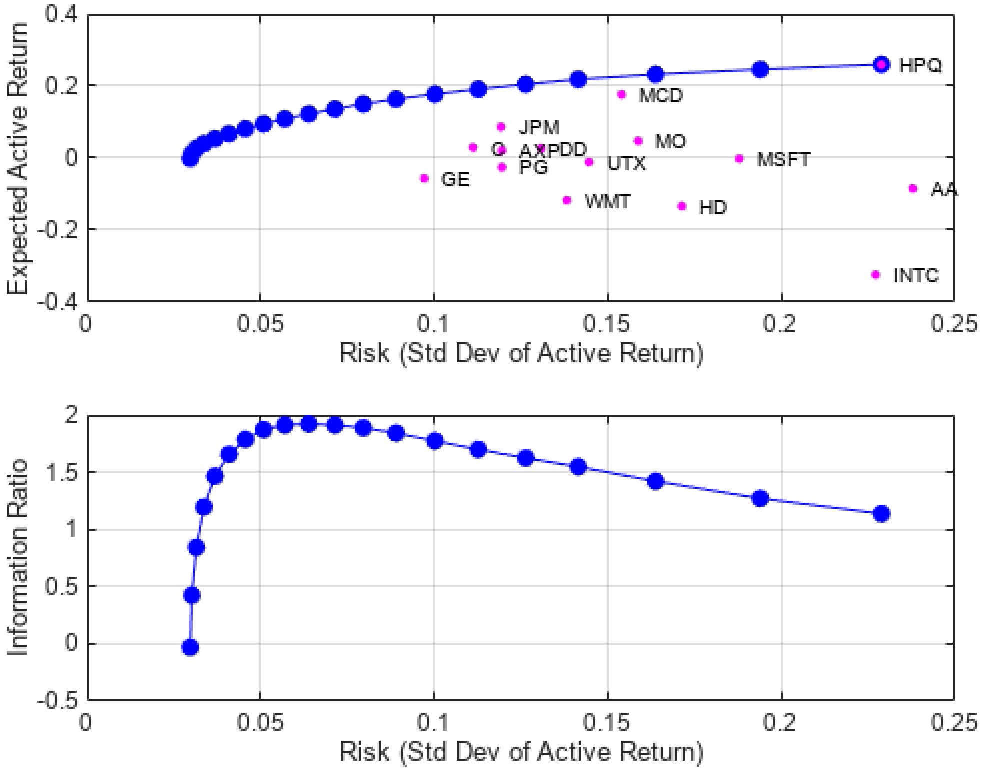
```
     TrackingPort: []
         Turnover: []
      BuyTurnover: []
     SellTurnover: []
             Name: []
        NumAssets: 15
        AssetList: {'AA'  'AXP'  'C'  'DD'  'GE'  'HD'  'HPQ'  'INTC'  'JPM'  'MCD'  'MO'  'MSFT
         InitPort: []
      AInequality: []
      bInequality: []
        AEquality: []
        bEquality: []
       LowerBound: [15x1 double]
       UpperBound: []
      LowerBudget: 1
      UpperBudget: 1
      GroupMatrix: []
       LowerGroup: []
       UpperGroup: []
           GroupA: []
           GroupB: []
       LowerRatio: []
       UpperRatio: []
     MinNumAssets: []
     MaxNumAssets: []
        BoundType: [15x1 categorical]
```

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.

```
pwgtAct = estimateFrontier(pAct, 20); % Estimate the weights.
[portRiskAct, portRetnAct] = estimatePortMoments(pAct, pwgtAct); % Get the risk and return.

% Extract the asset moments and names.
[assetActRetnDaily, assetActCovarDaily] = getAssetMoments(pAct);
assetActRiskDaily = sqrt(diag(assetActCovarDaily));
assetNames = pAct.AssetList;

% Rescale.
assetActRiskAnnual = sqrt(scale) * assetActRiskDaily;
portRiskAnnual  = sqrt(scale) *  portRiskAct;
assetActRetnAnnual = scale * assetActRetnDaily;
portRetnAnnual = scale *  portRetnAct;

figure;
subplot(2,1,1);
plot(portRiskAnnual, portRetnAnnual, 'bo-', 'MarkerFaceColor', 'b');
hold on;

scatter(assetActRiskAnnual, assetActRetnAnnual, 12, 'm', 'Filled');
hold on;
for k = 1:length(assetNames)
    text(assetActRiskAnnual(k) + 0.005, assetActRetnAnnual(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(portRiskAnnual, portRetnAnnual./portRiskAnnual, 'bo-', 'MarkerFaceColor', 'b');
xlabel('Risk (Std Dev of Active Return)');
ylabel('Information Ratio');
grid on;
```



Perform Information Ratio Maximization

Find the portfolio along the frontier with the maximum information ratio. The information ratio is the ratio of relative return to relative risk (also known as the "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. You can compute the information ratio using `estimateCustomObjectivePortfolio`.

```
infoRatio = @(x) (pAct.AssetMean'*x)/sqrt(x'*pAct.AssetCovar*x);
optWts = estimateCustomObjectivePortfolio(pAct,infoRatio,...
    ObjectiveSense="maximize");
```

Get the information ratio, risk, and return for the optimal portfolio.

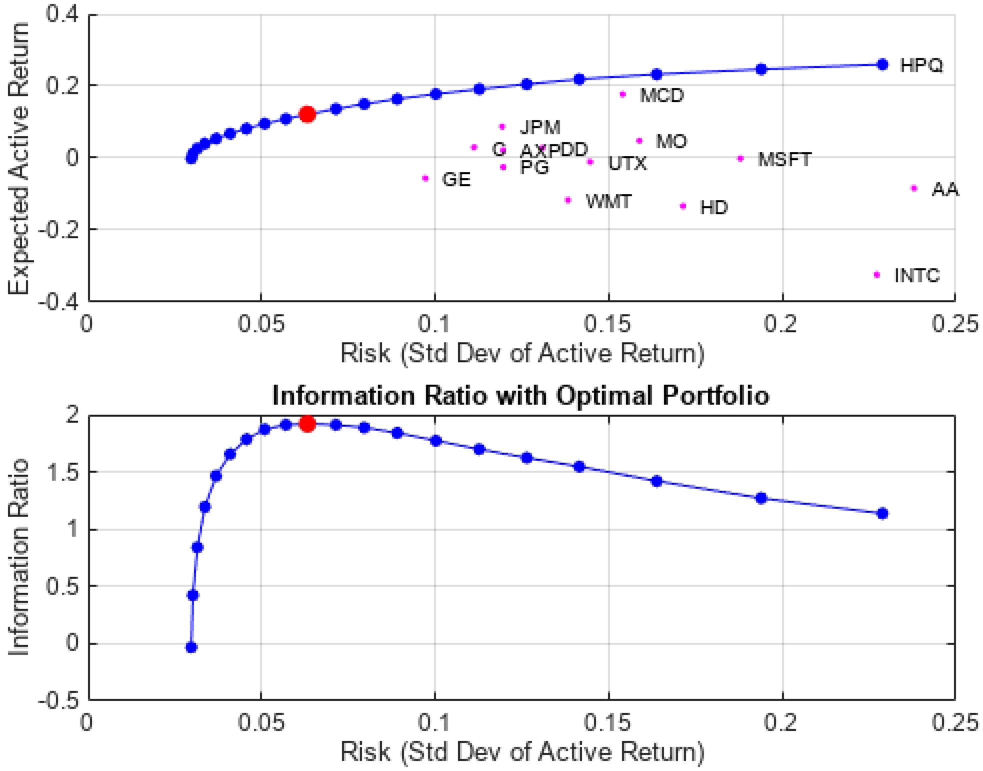
```
optInfoRatio = infoRatio(optWts);
[optPortRisk,optPortRetn] = estimatePortMoments(pAct,optWts)
```

```
optPortRisk = 0.0040

optPortRetn = 4.8166e-04
```

Plot the Optimal Portfolio

Verify that the portfolio found is indeed the maximum information-ratio portfolio.

```
% Rescale.
optPortRiskAnnual = sqrt(scale) * optPortRisk;
optPortReturnAnnual = scale * optPortRetn;

figure;
subplot(2,1,1);

scatter(assetActRiskAnnual, assetActRetnAnnual, 6, 'm', 'Filled');
hold on
for k = 1:length(assetNames)
    text(assetActRiskAnnual(k) + 0.005,assetActRetnAnnual(k),assetNames{k},'FontSize',8);
end
plot(portRiskAnnual,portRetnAnnual,'bo-','MarkerSize',4,'MarkerFaceColor','b');
plot(optPortRiskAnnual,optPortReturnAnnual,'ro-','MarkerFaceColor','r');
hold off;

xlabel('Risk (Std Dev of Active Return)');
ylabel('Expected Active Return');
grid on;

subplot(2,1,2);
plot(portRiskAnnual,portRetnAnnual./portRiskAnnual,'bo-','MarkerSize',4,'MarkerFaceColor','b');
hold on
plot(optPortRiskAnnual,optPortReturnAnnual./optPortRiskAnnual,'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', 'Weight
disp('Maximum Information Ratio Portfolio:')
```

```
Maximum Information Ratio Portfolio:
```

```
disp(results)
```

```
     Asset       Weight 
    ________    _______

    {'AA'  }     1.5389
    {'AXP' }    0.35545
    {'C'   }     9.6533
    {'DD'  }     4.0684
    {'HPQ' }     17.699
    {'JPM' }     21.565
    {'MCD' }     26.736
    {'MO'  }     13.648
    {'MSFT'}     2.6858
    {'UTX' }     2.0509
```

```
fprintf('Active return for Max. Info Ratio portfolio is %0.2f%%\n', optPortRetn*25200);
```

```
Active return for Max. Info Ratio portfolio is 12.14%
```

```
fprintf('Tracking error for Max. Info Ratio portfolio is %0.2f%%\n', optPortRisk*sqrt(252)*100);
```

```
Tracking error for Max. Info Ratio portfolio is 6.32%
```

See Also

Portfolio | inforatio | fminbnd

Related Examples

- “Creating the Portfolio Object” on page 4-24
- “Working with Portfolio Constraints Using Defaults” on page 4-57
- “Validate the Portfolio Problem for Portfolio Object” on page 4-90
- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-94
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-118
- “Postprocessing Results to Set Up Tradable Portfolios” on page 4-126
- “Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
- “Information Ratio” on page 7-7
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-183
- “Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215
- “Portfolio Optimization Using Factor Models” on page 4-224
- “Portfolio Optimization Using Social Performance Measure” on page 4-257
- “Diversify Portfolios Using Custom Objective” on page 4-329

More About

- “Performance Metrics Overview” on page 7-2
- “Portfolio Object” on page 4-19
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-17

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Portfolio Analysis with Turnover Constraints

This example shows how to analyze the characteristics of a portfolio of equities, and then compare them with the efficient frontier. This example seeks to answer the question of how much closer can you get to the efficient frontier by only risking a certain percentage of a portfolio to avoid transaction costs.

Import Data for the Portfolio Holdings

Load information on the current portfolio holdings from a Microsoft® Excel® spreadsheet into a table using the MATLAB® `readtable` function.

```
AssetHoldingData = readtable('portfolio.xls');
% Create a normalized current holdings vector that shows the respective
% investments as a percentage of total capital:
W = AssetHoldingData.Value/sum(AssetHoldingData.Value);
```

Import Market Data for Share Prices

Import the market data from a data source supported by Datafeed Toolbox™ that constitutes three years of closing prices for the stocks listed in the portfolio.

```
load SharePrices
```

Create a `Portfolio` Object

The `Portfolio` class enables you to use the imported data to create a `Portfolio` object. The `estimateAssetMoments` function for the `Portfolio` object enables you to set up a portfolio given only a historical price or returns series. The `estimateAssetMoments` function estimates mean and covariance of asset returns from data even if there is missing data.

```
P = Portfolio('Name', 'Sample Turnover Constraint Portfolio');
P = estimateAssetMoments(P,data,'DataFormat','Prices');

% You can assign text names to each asset in the portfolio.
P = setAssetList(P,AssetHoldingData.Symbol);

% Provide the current holdings.
P = setInitPort(P,W);
```

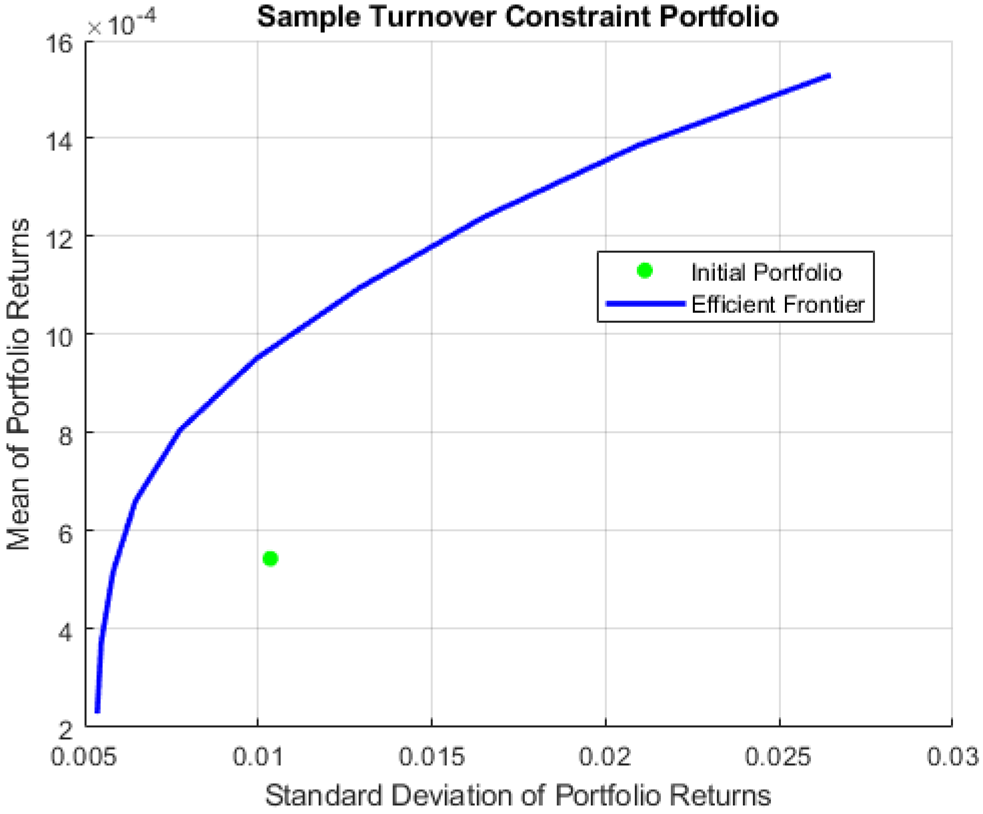
Perform Portfolio Optimization with No Turnover Constraint

The `Portfolio` object can optimize the holdings given any number of constraints. This example demonstrates using a simple, default constraint, that is, long positions only and 100% invested in assets.

```
P = setDefaultConstraints(P);
```

Visualize this efficient frontier with the `plotFrontier` function.

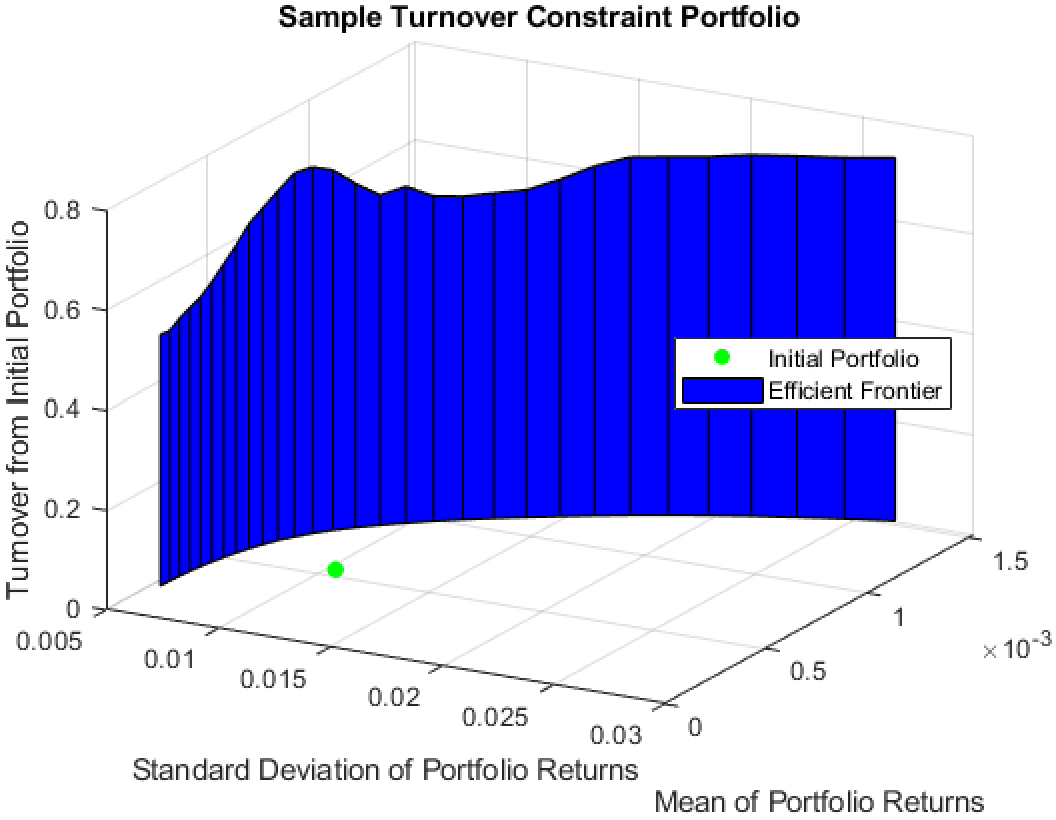
```
plotFrontier(P)
```



Visualize the Transaction Costs and Turnover

Due to transaction costs, it can be expensive to shift holdings from the current portfolio to a portfolio along this efficient frontier. The following custom plot shows that you must turn over between 50% and 75% of the holdings to get to this frontier.

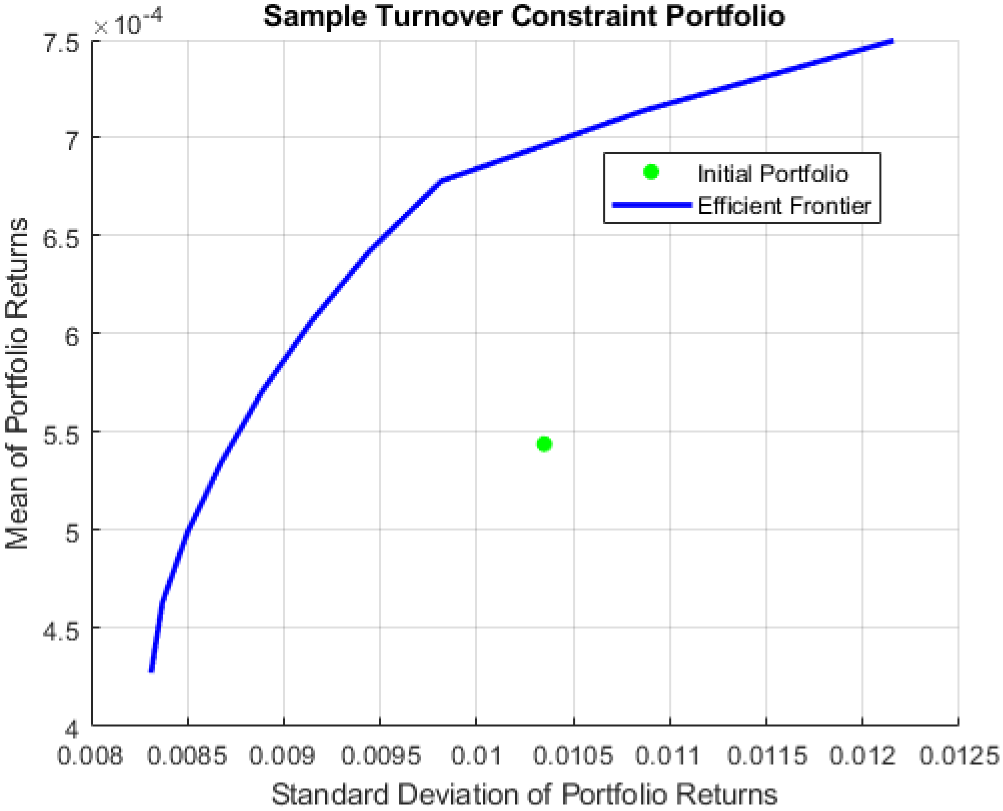
```
TurnoverPlot(P)
```



Perform Portfolio Optimization with a Turnover Constraint

How close can you get to this efficient frontier by only trading some of the portfolio? Assume that you want to trade only a certain percentage of the portfolio to avoid too much turnover in your holdings. This requirement imposes some nonlinear constraints on the problem and gives a problem with multiple local minima. Even so, the `Portfolio` object solves the problem, and you specify the turnover constraint using the `setTurnover` function.

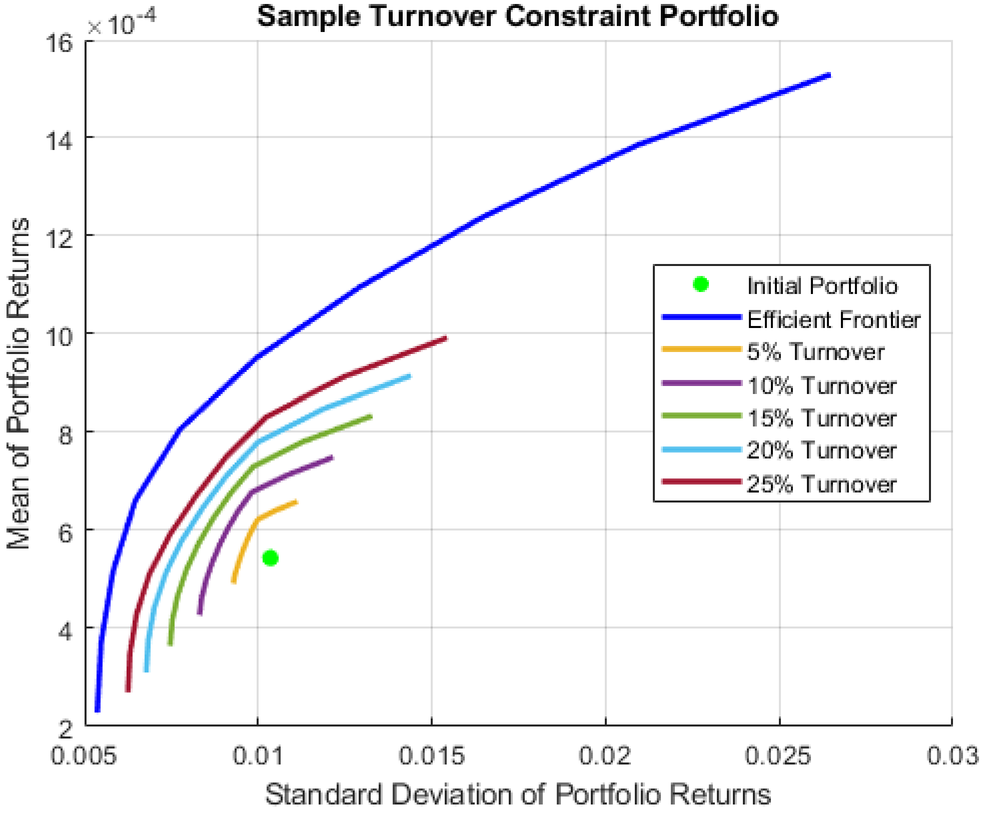
```
P10 = setTurnover(P,0.10);
plotFrontier(P10)
```



Visualize the Efficient Frontier at Different Turnover Thresholds

This efficient frontier is much closer to the initial portfolio than the starting efficient frontier without turnover constraints. To visualize this difference, use the custom function `TurnoverConstraintPlot` to visualize multiple constrained efficient frontiers at different turnover thresholds.

```
turnovers = 0.05:0.05:0.25;
TurnoverConstraintPlot(P,turnovers)
```



The `Portfolio` object is a powerful and efficient tool for performing various portfolio analysis tasks. In addition to turnover constraints, you can also optimize a `Portfolio` object for transaction costs for buying and selling portfolio assets using the `setCosts` function.

See Also

`Portfolio` | `setBounds` | `addGroups` | `setAssetMoments` | `estimateAssetMoments` | `estimateBounds` | `plotFrontier` | `estimateFrontierLimits` | `estimateFrontierByRisk` | `estimatePortRisk`

Related Examples

- “Creating the Portfolio Object” on page 4-24
- “Working with Portfolio Constraints Using Defaults” on page 4-57
- “Validate the Portfolio Problem for Portfolio Object” on page 4-90
- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-94
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-118
- “Postprocessing Results to Set Up Tradable Portfolios” on page 4-126
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-183
- “Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215
- “Portfolio Optimization Against a Benchmark” on page 4-195
- “Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152

- “Leverage in Portfolio Optimization with a Risk-Free Asset” on page 4-210
- “Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215
- “Portfolio Optimization Using Factor Models” on page 4-224
- “Portfolio Optimization Using Social Performance Measure” on page 4-257
- “Diversify Portfolios Using Custom Objective” on page 4-329

More About

- “Portfolio Object” on page 4-19
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-17

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Leverage in Portfolio Optimization with a Risk-Free Asset

This example shows how to use the `setBudget` function for the `Portfolio` class to define the limits on the `sum(AssetWeight_i)` in risky assets.

If the `sum(AssetWeight_i)` is less than `1`, the extra cash is invested in a risk-free asset. If the `sum(AssetWeight_i)` is larger than `1`, meaning that total risky asset investment is larger than initial available cash, the risk-free asset is shorted (borrowed) to fund the extra investment in a risky asset. The cost associated with borrowing a risk-free asset is automatically captured in the mean-variance optimization model for the `Portfolio` class. Therefore, you can use the `setBudget` function directly to control the level of leverage of cash for the portfolio.

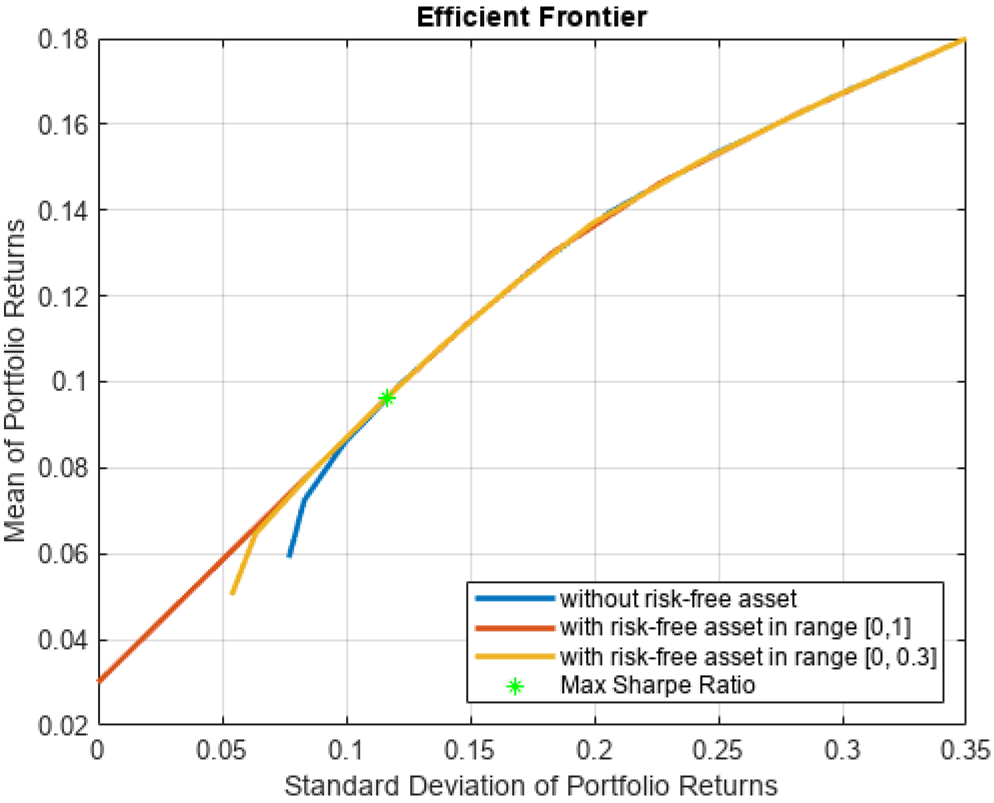
Portfolio Without Leverage

Consider the following example that does not leverage a risk-free asset.

```
assetsMean = [ 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];
riskFreeRate = 0.03;

% create a portfolio and define risk-free rate.
p = Portfolio('RiskFreeRate', riskFreeRate, 'assetmean', assetsMean, 'assetcovar', assetCovar, 'T
```

Create multiple portfolios with different budgets on risky assets to control the limits for investing in a risk-free asset.

```
p = setBudget(p, 1, 1);     % allow 0% risk free asset allocation, meaning fully invested in risk
p1 = setBudget(p, 0, 1);    % allow 0% to 100% risk free asset allocation
p2 = setBudget(p, 0.7, 1);  % allow 0% to 30% risk free asset allocation
[risk, retn] = estimatePortMoments(p, estimateMaxSharpeRatio(p));

figure;
plotFrontier(p); hold on;
plotFrontier(p1);hold on;
plotFrontier(p2); hold on;
plot(risk, retn, 'g*'); hold off;
legend('without risk-free asset', ...
    'with risk-free asset in range [0,1]', ...
    'with risk-free asset in range [0, 0.3]', ...
    'Max Sharpe Ratio', 'location','southeast');
```



In the efficient frontiers in the above figure, the lower-left part of the red efficient frontier line for the portfolio with a risk-free asset is in range `[0,1]` and is actually the capital allocation line (CAL). The slope of this line is the maximum Sharpe ratio of the portfolio, which demonstrates how return is best awarded by taking extra risk. The upper right of the red efficient frontier line is the same as a fully invested portfolio (blue line). Once the portfolio crosses the Sharpe ratio point, the portfolio is fully invested and there is no more cash available to allow high risk-awarded returns following the straight CAL. However, if borrowing of a risk-free asset is allowed, you can effectively use the funds from the borrowing of a risk-free asset to invest in more risky assets, as demonstrated in the "Portfolio with Leverage" section.

Portfolio with Leverage

To fund investments in risky assets, consider using leverage by borrowing a risk-free asset. The `Portfolio` class enables you to use leverage in asset allocation when a risk-free asset is available in the portfolio.

First, check if the `RiskFreeRate` property for the `Portfolio` object is nonzero.

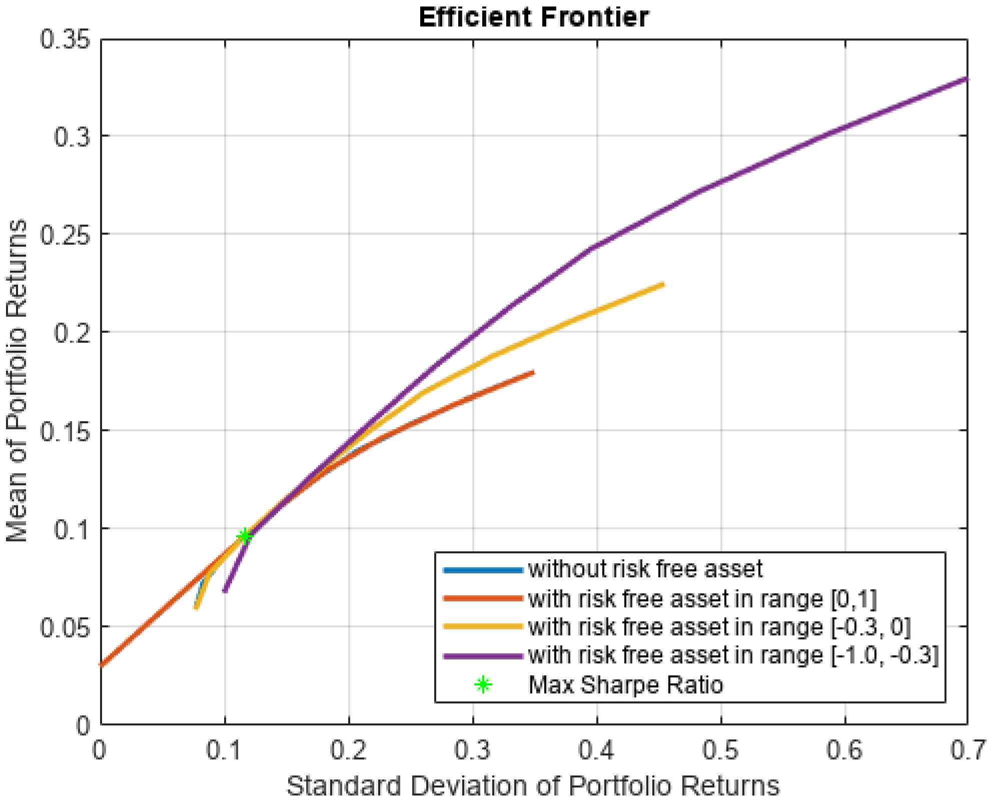
```
p
```

```
p = 
  Portfolio with properties:

          BuyCost: []
         SellCost: []
     RiskFreeRate: 0.0300
        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: []
     MinNumAssets: []
     MaxNumAssets: []
        BoundType: []
```

In this `Portfolio` object, the lower and upper budgets are both `1`. Limits must be set on the total investment in risky assets. Borrowing a risk-free asset funds the extra investment in risky assets. Use `setBudget` to set the lower and upper bounds to set the limits of the borrowed risk-free assets.

```
p = setBudget(p, 1, 1);      % allow 0% risk free asset allocation, meaning fully invested in ris
p3 = setBudget(p, 1, 1.3);   % allow 0% risk free asset allocation, and allow borrowing of risk
p4 = setBudget(p, 1.3, 2);   % allow 0% risk free asset allocation, and allow borrowing of risk
[risk, retn] = estimatePortMoments(p, estimateMaxSharpeRatio(p));

figure
plotFrontier(p); hold on;
plotFrontier(p1);hold on;
plotFrontier(p3); hold on;
plotFrontier(p4); hold on;
plot(risk, retn, 'g*'); hold off;
legend('without risk free asset',  ...
    'with risk free asset in range [0,1]', ...
    'with risk free asset in range [-0.3, 0]', ...
    'with risk free asset in range [-1.0, -0.3]',...
    'Max Sharpe Ratio', 'location','southeast');
```



In this figure, the upper-right parts of both the orange and purple efficient frontiers extend from the CAL (lower-left red line), because of the leverage of a risk-free asset. The same levels of risk-awarded returns are obtained. Once the portfolio exhausts the maximum allowed leverage, the efficient frontier starts to fall below the CAL again, resulting in portfolios with lower Sharpe ratios.

See Also

`Portfolio` | `setBounds` | `addGroups` | `setAssetMoments` | `estimateAssetMoments` | `estimateBounds` | `plotFrontier` | `estimateFrontierLimits` | `estimateFrontierByRisk` | `estimatePortRisk`

Related Examples

- “Creating the Portfolio Object” on page 4-24
- “Working with Portfolio Constraints Using Defaults” on page 4-57
- “Validate the Portfolio Problem for Portfolio Object” on page 4-90
- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-94
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-118
- “Postprocessing Results to Set Up Tradable Portfolios” on page 4-126
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-183
- “Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215
- “Portfolio Optimization Against a Benchmark” on page 4-195

- “Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
- “Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215
- “Portfolio Optimization Using Factor Models” on page 4-224
- “Portfolio Optimization Using Social Performance Measure” on page 4-257
- “Diversify Portfolios Using Custom Objective” on page 4-329

More About

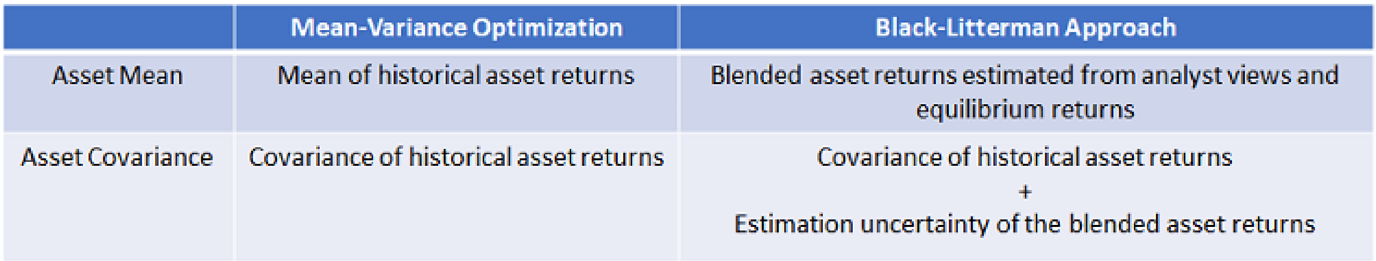
- “Portfolio Object” on page 4-19
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-17

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Black-Litterman Portfolio Optimization Using Financial Toolbox™

This example shows the workflow to implement the Black-Litterman model with the `Portfolio` class in Financial Toolbox™. The Black-Litterman model is an asset allocation approach that allows investment analysts to incorporate subjective views (based on investment analyst estimates) into market equilibrium returns. By blending analyst views and equilibrium returns instead of relying only on historical asset returns, the Black-Litterman model provides a systematic way to estimate the mean and covariance of asset returns.



	Mean-Variance Optimization	Black-Litterman Approach
Asset Mean	Mean of historical asset returns	Blended asset returns estimated from analyst views and equilibrium returns
Asset Covariance	Covariance of historical asset returns	Covariance of historical asset returns + Estimation uncertainty of the blended asset returns

In the Black-Litterman model, the blended expected return is $\bar{\mu} = \left[P^T\Omega^{-1}P + C^{-1}\right]^{-1}\left[P^T\Omega^{-1}q + C^{-1}\pi\right]$ and the estimation uncertainty is $\mathrm{cov}(\mu) = \left[P^T\Omega^{-1}P + C^{-1}\right]^{-1}$. To use the Black-Litterman model, you must prepare the inputs: P, q , Ω , π, and C. The inputs for P, q , and Ω are view-related and defined by the investment analyst. π is the equilibrium return and C is the uncertainty in prior belief. This example guides you to define these inputs and use the resulting blended returns in a portfolio optimization. For more information on the concept and derivation of the Black-Litterman model, see the Appendix section Black-Litterman Model under a Bayesian Framework on page 4-221.

Define the Universe of Assets

The `dowPortfolio.xlsx` data set includes 30 assets and one benchmark. Seven assets from this data set comprise the investment universe in this example. The risk-free rate is assumed to be zero.

```
T = readtable('dowPortfolio.xlsx');
```

Define the asset universe and extract the asset returns from the price data.

```
assetNames = ["AA", "AIG", "WMT", "MSFT", "BA", "GE", "IBM"];
benchmarkName = "DJI";
head(T(:,["Dates" benchmarkName assetNames]))
```

```
       Dates        DJI       AA       AIG      WMT      MSFT      BA       GE       IBM 
    ___________    _____    _____    _____    _____    _____    _____    _____    _____

    03-Jan-2006    10847    28.72    68.41     44.9    26.19    68.63     33.6    80.13
    04-Jan-2006    10880    28.89    68.51    44.99    26.32    69.34    33.56    80.03
    05-Jan-2006    10882    29.12     68.6    44.38    26.34    68.53    33.47    80.56
    06-Jan-2006    10959    29.02    68.89    44.56    26.26    67.57     33.7    82.96
    09-Jan-2006    11012    29.37    68.57     44.4    26.21    67.01    33.61    81.76
    10-Jan-2006    11012    28.44    69.18    44.54    26.35    67.33    33.43     82.1
    11-Jan-2006    11043    28.05     69.6    45.23    26.63     68.3    33.66    82.19
    12-Jan-2006    10962    27.68    69.04    44.43    26.48     67.9    33.25    81.61
```

```
retnsT = tick2ret(T(:, 2:end));
assetRetns = retnsT(:, assetNames);
benchRetn = retnsT(:, "DJI");
numAssets = size(assetRetns, 2);
```

Specify Views of the Market

The views represent the subjective views of the investment analyst regarding future market changes, expressed as $\boldsymbol{q} = \boldsymbol{P} * \mu + \varepsilon, \;\; \varepsilon \sim N(0, \Omega), \; \Omega = \mathrm{diag}(\omega_1, \omega_2, \ldots \omega_v)$, where v is total number of views. For more information, see the Appendix section Assumptions and Views on page 4-221. With v views and k assets, P is a v-by-k matrix, $\boldsymbol{q}$ is a v-by-1 vector, and Ω is a v-by-v diagonal matrix (representing the independent uncertainty in the views). The views do not necessarily need to be independent among themselves and the structure of Ω can be chosen to account for investment analyst uncertainties in the views [4 on page 4-222]. The smaller the ω_i in Ω, the smaller the variance in the distribution of the ith view, and the stronger or more certain the investor's ith view. This example assumes three independent views.

1. AIG is going to have 5% annual return with uncertainty 1e-3. This is a weak absolute view due to its high uncertainty.
2. WMT is going to have 3% annual return with uncertainty 1e-3. This is a weak absolute view due to its high uncertainty.
3. MSFT is going to outperform IBM by 5% annual return with uncertainty 1e-5. This is a strong relative view due to its low uncertainty.

```
v = 3;  % total 3 views
P = zeros(v, numAssets);
q = zeros(v, 1);
Omega = zeros(v);

% View 1
P(1, assetNames=="AIG") = 1;
q(1) = 0.05;
Omega(1, 1) = 1e-3;

% View 2
P(2, assetNames=="WMT") = 1;
q(2) = 0.03;
Omega(2, 2) = 1e-3;

% View 3
P(3, assetNames=="MSFT") = 1;
P(3, assetNames=="IBM") = -1;
q(3) = 0.05;
Omega(3, 3) = 1e-5;
```

Visualize the three views in table form.

```
viewTable = array2table([P q diag(Omega)], 'VariableNames', [assetNames "View_Return" "View_Unce
```

```
viewTable=3×9 table
    AA    AIG    WMT    MSFT    BA    GE    IBM    View_Return    View_Uncertainty
    __    ___    ___    ____    __    __    ___    ___________    ________________

    0      1      0      0      0     0      0        0.05              0.001     
    0      0      1      0      0     0      0        0.03              0.001     
    0      0      0      1      0     0     -1        0.05              1e-05     
```

Because the returns from `dowPortfolio.xlsx` data set are daily returns and the views are on the annual returns, you must convert views to be on daily returns.

```
bizyear2bizday = 1/252;
q = q*bizyear2bizday;
Omega = Omega*bizyear2bizday;
```

Estimate the Covariance from the Historical Asset Returns

Σ is the covariance of the historical asset returns.

```
Sigma = cov(assetRetns.Variables);
```

Define the Uncertainty *C*

The Black-Litterman model makes the assumption that the structure of C is proportional to the covariance Σ. Therefore, $C = \tau\Sigma$, where τ is a small constant. A smaller τ indicates a higher confidence in the prior belief of μ. The work of He and Litterman uses a value of 0.025. Other authors suggest using $1/n$ where n is the number of data points used to generate the covariance matrix [3 on page 4-222]. This example uses $1/n$.

```
tau = 1/size(assetRetns.Variables, 1);
C = tau*Sigma;
```

Market Implied Equilibrium Return

In the absence of any views, the equilibrium returns are likely equal to the implied returns from the equilibrium portfolio holding. In practice, the applicable equilibrium portfolio holding can be any optimal portfolio that the investment analyst would use in the absence of additional views on the market, such as the portfolio benchmark, an index, or even the current portfolio [2 on page 4-222]. In this example, you use linear regression to find a market portfolio that tracks the returns of the DJI benchmark. Then, you use the market portfolio as the equilibrium portfolio and the equilibrium returns are implied from the market portfolio. The `findMarketPortfolioAndImpliedReturn` function, defined in Local Functions on page 4-220, implements the equilibrium returns. This function takes historical asset returns and benchmark returns as inputs and outputs the market portfolio and the corresponding implied returns.

```
[wtsMarket, PI] = findMarketPortfolioAndImpliedReturn(assetRetns.Variables, benchRetn.Variables)
```

Compute the Estimated Mean Return and Covariance

Use the P, q , Ω , π , and C inputs to compute the blended asset return and variance using the Black-Litterman model.

You can compute $\bar{\mu}$ and $\mathrm{cov}(\mu)$ directly by using this matrix operation:

$$\bar{\mu} = \left[P^T\Omega^{-1}P + C^{-1}\right]^{-1}\left[P^T\Omega^{-1}q + C^{-1}\pi\right], \mathrm{cov}(\mu) = \left[P^T\Omega^{-1}P + C^{-1}\right]^{-1}$$

```
mu_bl = (P'*(Omega\P) + inv(C)) \ ( C\PI + P'*(Omega\q));
cov_mu = inv(P'*(Omega\P) + inv(C));
```

Comparing the blended expected return from Black-Litterman model to the prior belief of expected return π, you find that the expected return from Black-Litterman model is indeed a mixture of both prior belief and investor views. For example, as shown in the table below, the prior belief assumes similar returns for MSFT and IBM, but in the blended expected return, MSFT has a higher return than IBM by more than 4%. This difference is due to the imposed strong view that MSFT outperforms IBM by 5%.

```
table(assetNames', PI*252, mu_bl*252, 'VariableNames', ["Asset_Name", ...
    "Prior_Belief_of_Expected_Return", "Black_Litterman_Blended_Expected_Return"])
```

```
ans=7×3 table
    Asset_Name    Prior_Belief_of_Expected_Return    Black_Litterman_Blended_Expected_Return
    __________    _______________________________    _______________________________________

      "AA"                    0.19143                                0.19012                
      "AIG"                   0.14432                                0.13303                
      "WMT"                   0.15754                                 0.1408                
      "MSFT"                  0.14071                                0.17557                
      "BA"                    0.21108                                 0.2017                
      "GE"                    0.13323                                0.12525                
      "IBM"                   0.14816                                0.12877                
```

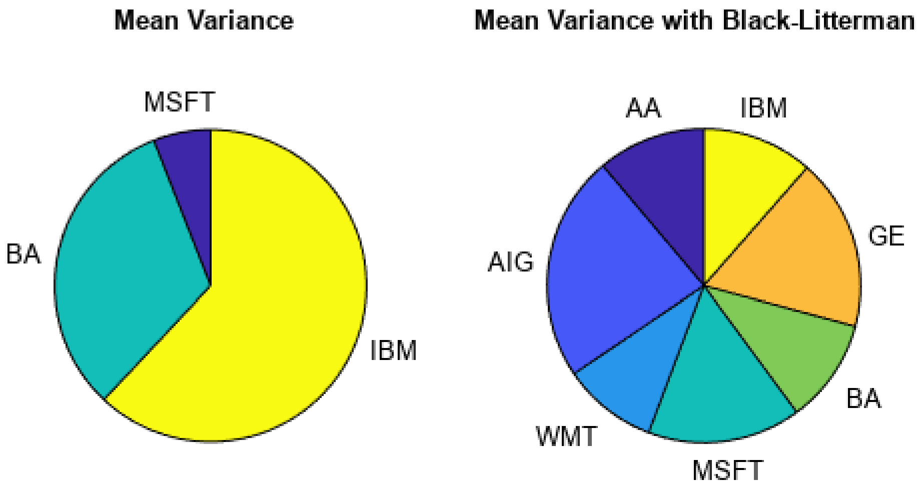
Portfolio Optimization and Results

The `Portfolio` object in Financial Toolbox™ implements the Markowitz mean variance portfolio optimization framework. Using a `Portfolio` object, you can find the efficient portfolio for a given risk or return level, and you can also maximize the Sharpe ratio.

Use `estimateMaxSharpeRatio` with the `Portfolio` object to find allocations with the maximum Sharpe ratio for the following portfolios:

- Portfolio with asset mean and covariance from historical asset returns
- Portfolio with blended asset return and covariance from the Black-Litterman model

```
port = Portfolio('NumAssets', numAssets, 'lb', 0, 'budget', 1, 'Name', 'Mean Variance');
port = setAssetMoments(port, mean(assetRetns.Variables), Sigma);
wts = estimateMaxSharpeRatio(port);

portBL = Portfolio('NumAssets', numAssets, 'lb', 0, 'budget', 1, 'Name', 'Mean Variance with Blac
portBL = setAssetMoments(portBL, mu_bl, Sigma + cov_mu);
wtsBL = estimateMaxSharpeRatio(portBL);

ax1 = subplot(1,2,1);
idx = wts>0.001;
pie(ax1, wts(idx), assetNames(idx));
title(ax1, port.Name ,'Position', [-0.05, 1.6, 0]);

ax2 = subplot(1,2,2);
idx_BL = wtsBL>0.001;
pie(ax2, wtsBL(idx_BL), assetNames(idx_BL));
title(ax2, portBL.Name ,'Position', [-0.05, 1.6, 0]);
```



```
table(assetNames', wts, wtsBL, 'VariableNames', ["AssetName", "Mean_Variance", ...
     "Mean_Variance_with_Black_Litterman"])
```

```
ans=7×3 table
    AssetName    Mean_Variance    Mean_Variance_with_Black_Litterman
    _________    _____________    __________________________________

     "AA"         6.6352e-16                    0.1115              
     "AIG"        5.7712e-17                   0.23314              
     "WMT"        1.8628e-17                  0.098048              
     "MSFT"         0.059393                   0.15824              
     "BA"            0.32068                   0.10748              
     "GE"         7.7553e-15                    0.1772              
     "IBM"           0.61993                   0.11439              
```

When you use the values for the blended asset return and the covariance from the Black-Litterman model in a mean-variance optimization, the optimal allocations reflect the views of the investment analyst directly. The allocation from the Black-Litterman model is more diversified, as the pie chart shows. Also, the weights among the assets in the Black-Litterman model agree with the investment analyst views. For example, when you compare the Black-Litterman result with the plain mean-variance optimization result, you can see that the Black-Litterman result is more heavily invested in MSFT than in IBM. This is because the investment analyst has a strong view that MSFT will outperform IBM.

Local Functions

```
function [wtsMarket, PI] = findMarketPortfolioAndImpliedReturn(assetRetn, benchRetn)
% Find the market portfolio that tracks the benchmark and its corresponding implied expected retu
```

The implied return is calculated by reverse optimization. The risk-free rate is assumed to be zero. The general formulation of a portfolio optimization is given by the Markowitz optimization problem: $\underset{\omega}{\operatorname{argmax}}\ \omega^T\mu - \frac{\delta}{2}\omega^T\Sigma\omega$. Here ω is an N-element vector of asset weights, μ is an N-element vector of expected asset returns, Σ is the N-by-N covariance matrix of asset returns, and δ is a positive risk aversion parameter. Given δ , in the absence of constraints, a closed form solution to this problem is $\omega = \frac{1}{\delta}\Sigma^{-1}\mu$. Therefore, with a market portfolio, the implied expected return is $\pi = \delta\Sigma\omega_{mkt}$.

To compute an implied expected return, you need $\Sigma,\ \ \omega_{mkt},\ \ \delta$.

1) Find Σ.

Σ is calculated from historical asset returns.

```
Sigma = cov(assetRetn);
```

2) Find the market portfolio.

To find the market portfolio, regress against the DJI. The imposed constraints are fully invested and long only: $\sum_{i=1}^{n}\omega_i = 1, 0 \le \omega_i,\ \ \forall i \in \{1, . . . , n\}$

```
numAssets = size(assetRetn,2);
LB = zeros(1,numAssets);
Aeq = ones(1,numAssets);
Beq = 1;
opts = optimoptions('lsqlin','Algorithm','interior-point', 'Display',"off");
wtsMarket = lsqlin(assetRetn, benchRetn, [], [], Aeq, Beq, LB, [], [], opts);
```

3) Find δ.

Multiply both sides of $\pi = \delta\Sigma\omega_{mkt}$ with ω_{mkt}^T to output $\delta = \frac{SharpeRatio}{\sigma_m}$. Here, the Benchmark is assumed to be maximizing the Sharpe ratio and the corresponding value is used as market Sharpe ratio. Alternatively, you can calibrate an annualized Sharpe ratio to be 0.5, which leads to `shpr`=0.5/`sqrt`(252) [1 on page 4-222]. σ_m is the standard deviation of the market portfolio.

```
shpr = mean(benchRetn)/std(benchRetn);
delta = shpr/sqrt(wtsMarket'*Sigma*wtsMarket);
```

4) Compute the implied expected return.

Assuming that the market portfolio maximizes the Sharpe ratio, the implied return, without the effects from constraints, is computed directly as $\pi = \delta\Sigma\omega$.

```
PI = delta*Sigma*wtsMarket;
end
```

Appendix: Black-Litterman Model Under a Bayesian Framework

Assumptions and Views

Assume that the investment universe is composed of k assets and the vector of asset returns $\boldsymbol{r}$ is modeled as a random variable, following a multivariate normal distribution $\boldsymbol{r} \sim N(\mu, \Sigma)$. Σ is the covariance from historical asset returns. The unknown model parameter is the expected return μ. From the perspective of Bayesian statistics, the Black-Litterman model attempts to estimate μ by combining the investment analyst views (or "observations of the future") and some prior knowledge about μ.

In addition, assume the prior knowledge that μ is a normally distributed random variable $\mu \sim N(\pi, \boldsymbol{C})$ [1, 2 on page 4-222]. In the absence of any views (observations), the prior mean π is likely to be the equilibrium returns, implied from the equilibrium portfolio holding. In practice, the applicable equilibrium portfolio holding is not necessarily the equilibrium portfolio, but rather a target optimal portfolio that the investment analyst would use in the absence of additional views on the market, such as the portfolio benchmark, an index, or even the current portfolio. C represents the uncertainty in the prior and the Black-Litterman model makes the assumption that the structure of C is $\tau\Sigma$. τ is a small constant, and many authors use different values. A detailed discussion about τ can be found in [3 on page 4-222].

Observations are necessary to perform a statistical inference on μ. In the Black-Litterman model, the observations are views about future asset returns expressed at the portfolio level. A view is the expected return of a portfolio composed of the universe of k assets. Usually, the portfolio return has uncertainty, so an error term is added to catch the departure. Assume that there is a total of v views. For a view i, $\boldsymbol{p}_i$ is a row vector with dimension 1 x k, and q_i is a scalar [2 on page 4-222].

$$q_i = E[\boldsymbol{p}_i * \boldsymbol{r} \mid \mu] + \varepsilon_i\ ,\ \ i = 1, 2, \ldots, v$$

You can stack the v views vertically, and Ω is the covariance of the uncertainties from all views. Assume that the uncertainties are independent.

$$\boldsymbol{q}\ =\ E[\boldsymbol{P} * \boldsymbol{r} \mid \mu] +\ \varepsilon,\ \ \varepsilon \sim N(0,\ \Omega),\ \ \Omega = \text{diag}(\omega_1, \omega_2, \ldots \omega_v).$$

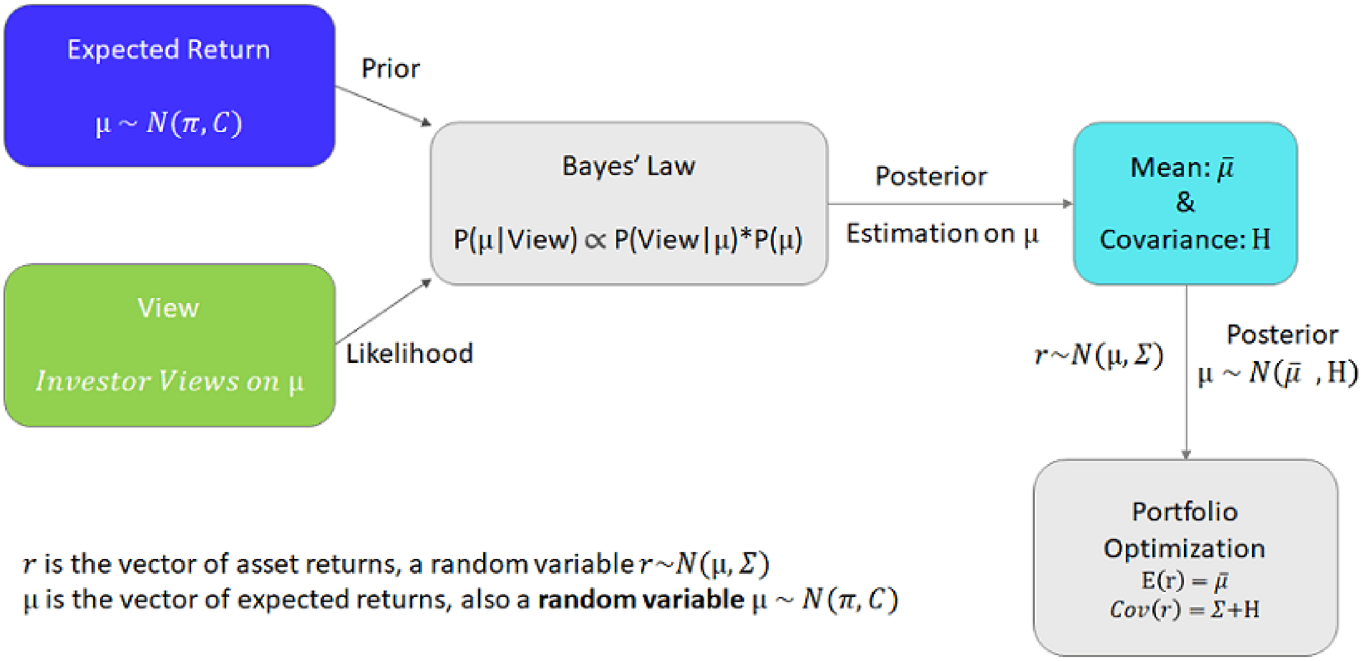
Note that Ω does not necessarily need to be a diagonal matrix. The investment analyst can choose the structure of Ω to account for their uncertainties in the views [4 on page 4-222].

Under the previous assumption $\boldsymbol{r} \sim N(\mu, \Sigma)$, it follows that

$$\boldsymbol{q}\ = \boldsymbol{P} * \mu\ +\ \varepsilon,\ \ \varepsilon \sim N(0,\ \Omega),\ \Omega = \text{diag}(\omega_1, \omega_2, \ldots \omega_v).$$

The Bayesian Definition of the Black-Litterman Model

Based on Bayesian statistics, it is known that: posterior $\propto$ likelihood $*$ prior.



r is the vector of asset returns, a random variable $r \sim N(\mu, \Sigma)$
μ is the vector of expected returns, also a **random variable** $\mu \sim N(\pi, C)$

Figure 1. Black-Litterman Model

In the context of the Black-Litterman model, posterior ∝ likelihood * prior is expressed as $f(\mu|q) \propto f(q|\mu)*f(\mu)$, where each Bayesian term is defined as follows [2 on page 4-222]:

- The *likelihood* is how likely it is for the views to happen given μ and is expressed as $f(q|\mu) \propto \exp\left[-\frac{1}{2}(P\mu - q)'\Omega^{-1}(P\mu - q)\right]$.
- The *prior* assumes the prior knowledge that $\mu \sim N(\pi, C)$ and is expressed as $f(\mu) \propto \exp\left[-\frac{1}{2}(\mu - \pi)'C^{-1}(\mu - \pi)\right]$.
- The *posterior* is the distribution of μ given views and is expressed as $f(\mu|q) \propto \exp\left[-\frac{1}{2}(P\mu - q)'\Omega^{-1}(P\mu - q) - \frac{1}{2}(\mu - \pi)'C^{-1}(\mu - \pi)\right]$.

As previously stated, the posterior distribution of μ is also a normal distribution. By completing the squares, you can derive the posterior mean and covariance as $\bar{\mu} = \left[P^T\Omega^{-1}P + C^{-1}\right]^{-1}\left[P^T\Omega^{-1}q + C^{-1}\pi\right]$, $\mathrm{cov}(\mu) = \left[P^T\Omega^{-1}P + C^{-1}\right]^{-1}$.

Finally, by combining the Bayesian posterior distribution of μ and the model of asset returns $\boldsymbol{r} \sim N(\mu, \Sigma)$, you then have the posterior prediction of asset returns as $\boldsymbol{r} \sim N(\bar{\mu},\ \Sigma + \mathrm{cov}(\mu))$.

References

1 Walters, J. "The Black-Litterman Model in Detail." 2014. Available at SSRN: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=1314585.

2 Kolm, P. N., and Ritter, G. "On the Bayesian Interpretation of Black-Litterman." *European Journal of Operational Research*. Vol. 258, Number 2, 2017, pp. 564-572.

3 Attilio, M. "Beyond Black-Litterman in Practice: A Five-Step Recipe to Input Views on Non-Normal Markets." 2006. Available at SSRN: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=872577.

4 Ulf, H. "Computing implied returns in a meaningful way." *Journal of Asset Management.* Vol 6, Number 1, 2005, pp. 53-64.

See Also

`Portfolio` | `setBounds` | `addGroups` | `setAssetMoments` | `estimateAssetMoments` | `estimateBounds` | `plotFrontier` | `estimateFrontierLimits` | `estimateFrontierByRisk` | `estimatePortRisk`

Related Examples

- “Creating the Portfolio Object” on page 4-24
- “Working with Portfolio Constraints Using Defaults” on page 4-57
- “Validate the Portfolio Problem for Portfolio Object” on page 4-90
- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-94
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-118
- “Postprocessing Results to Set Up Tradable Portfolios” on page 4-126
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-183
- “Portfolio Optimization Against a Benchmark” on page 4-195
- “Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
- “Portfolio Optimization Using Factor Models” on page 4-224
- “Portfolio Optimization Using Social Performance Measure” on page 4-257
- “Diversify Portfolios Using Custom Objective” on page 4-329

More About

- “Portfolio Object” on page 4-19
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-17

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Portfolio Optimization Using Factor Models

This example shows two approaches for using a factor model to optimize asset allocation under a mean-variance framework. Multifactor models are often used in risk modeling, portfolio management, and portfolio performance attribution. A multifactor model reduces the dimension of the investment universe and is responsible for describing most of the randomness of the market [1 on page 4-229]. The factors can be statistical, macroeconomic, and fundamental. In the first approach in this example, you build statistical factors from asset returns and optimize the allocation directly against the factors. In the second approach you use the given factor information to compute the covariance matrix of the asset returns and then use the `Portfolio` class to optimize the asset allocation.

Load Data

Load a simulated data set, which includes asset returns total $p = 100$ assets and 2000 daily observations.

```
load('asset_return_100_simulated.mat');
[nObservation, p] = size(stockReturns)
```

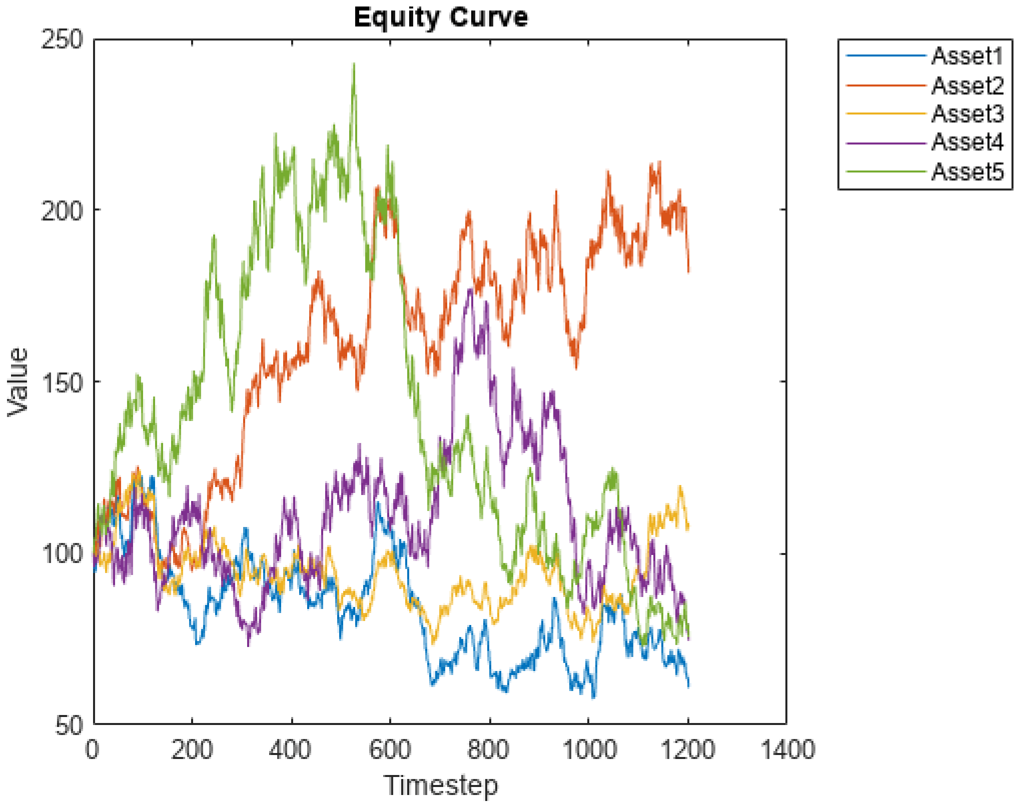
```
nObservation = 2000
```

```
p = 100
```

```
splitPoint = ceil(nObservation*0.6);
training = 1:splitPoint;
test = splitPoint+1:nObservation;
trainingNum = numel(training);
```

Visualize the equity curve for each stock. For this example, plot the first five stocks.

```
plot(ret2tick(stockReturns{training,1:5}, 'method', 'simple')*100); hold off;
xlabel('Timestep');
ylabel('Value');
title('Equity Curve');
legend(stockReturns.Properties.VariableNames(1:5), 'Location',"bestoutside",'Interpreter', 'none
```



Optimize the Asset Allocation Directly Against Factors with a Problem-Based Definition Framework

For the factors, you can use statistical factors extracted from the asset return series. In this example, you use principal component analysis (PCA) to extract statistical factors [1 on page 4-229]. You can then use this factor model to solve the portfolio optimization problem.

With a factor model, p asset returns can be expressed as a linear combination of k factor returns, $r_a = \mu_a + F\, r_f + \varepsilon_a$, where $k << p$. In the mean-variance framework, portfolio risk is

$$\mathrm{Var}(R_p) = \mathrm{Var}\left(r_a{}^T w_a\right) = \mathrm{Var}\left(\left(\mu_a + F\, r_f + \varepsilon_a\right)^T w_a\right) = w_a^T\left(F \Sigma_f F^T + D\right) w_a = w_f^T\, \Sigma_f\, w_f + w_a^T\, D\, w_a, \text{ with}$$
$$w_f = F^T w_a,$$

where:

R_p is the portfolio return (a scalar).

r_a is the asset return.

μ_a is the mean of asset return.

F is the factor loading, with the dimensions *p-by-k*.

r_f is the factor return.

ε_a is the idiosyncratic return related to each asset.

w_a is the asset weight.

w_f is the factor weight.

Σ_f is the covariance of factor returns.

D is the variance of idiosyncratic returns.

The parameters r_a, w_a, μ_a, ε_a are p-by-1 column vectors, $r_f$and w_f are k-by-1 column vectors, Σ_a is a p-by-p matrix, Σ_k is a k-by-k matrix, and D is a p-by-p diagonal matrix.

Therefore, the mean-variance optimization problem is formulated as

$$\max \mu_a^T w_a\,, \;\; s.t.\, F^T w_a = w_f,\, w_f^T \Sigma_f w_f + w_a^T D\, w_a \le t_{\text{risk}}, 0 \le w \le 1,\, e^T w_a = 1.$$

In the p dimensional space formed by p asset returns, PCA finds the most important k directions that capture the most important variations in the given returns of p assets. Usually, k is less than p. Therefore, by using PCA you can decompose the p asset returns into the k factors, which greatly reduces the dimension of the problem. The k principal directions are interpreted as factor loadings and the scores from the decomposition are interpreted as factor returns. For more information, see `pca` (Statistics and Machine Learning Toolbox™). In this example, use k = `10` as the number of principal components. Alternatively, you can also find the k value by defining a threshold for the total variance represented as the top k principal components. Usually 95% is an acceptable threshold.

```
k = 10;
[factorLoading,factorRetn,latent,tsq,explained,mu] = pca(stockReturns{training,:}, 'NumComponents
disp(size(factorLoading))
```

```
   100    10
```

```
disp(size(factorRetn))
```

```
        1200           10
```

In the output p-by-k `factorLoading`, each column is a principal component. The asset return vector at each timestep is decomposed to these k dimensional spaces, where $k << p$. The output `factorRetn` is a *trainingNum*-by-k dimension.

Alternatively, you can also use the `covarianceDenoising` function to estimate a covariance matrix (`denoisedCov`) using denoising to reduce the noise and enhance the signal of the empirical covariance matrix. [1 on page 4-229] The goal of denoising is to separate the eigenvalues associated with signal from the ones associated with noise and shrink the eigenvalues associated with noise to enhance the signal. Thus, `covarianceDenoising` is a reasonable tool to identify the number of factors.

```
[denoisedCov,numFactorsDenoising] = covarianceDenoising(stockReturns{training,:});
```

The number of factors (`numFactorsDenoising`) is `6`.

```
disp(numFactorsDenoising)
```

```
     6
```

Estimate the factor covariance matrix with factor returns (`factorRetn`) obtained using the `pca` function.

```
covarFactor = cov(factorRetn);
```

You can reconstruct the p asset returns for each observation using each k factor returns by following $r_a = \mu_a + F\, r_f + \varepsilon_a$.

Reconstruct the total 1200 observations for the training set.

```
reconReturn = factorRetn*factorLoading' + mu;
unexplainedRetn = stockReturns{training,:} - reconReturn;
```

There are unexplained asset returns ε_a because the remaining (p - k) principal components are dropped. You can attribute the unexplained asset returns to the asset-specific risks represented as D.

```
unexplainedCovar = diag(cov(unexplainedRetn));
D = diag(unexplainedCovar);
```

You can use a problem-based definition framework from Optimization Toolbox™ to construct the variables, objective, and constraints for the problem: max $\mu_a^T\, w_a\, ,\ s.t.\ F^T w_a = w_f$, $w_f^T\, \Sigma_f\, w_f + w_a^T\, D\, w_a \le t_{\text{risk}}, e^T w_a = 1, 0 \le w \le 1$. The problem-based definition framework enables you to define variables and express objective and constraints symbolically. You can add other constraints or use a different objective based on your specific problem. For more information, see “First Choose Problem-Based or Solver-Based Approach”.

```
targetRisk = 0.007;  % Standard deviation of portfolio return
tRisk = targetRisk*targetRisk;  % Variance of portfolio return
meanStockRetn = mean(stockReturns{training,:});

optimProb = optimproblem('Description','Portfolio with factor covariance matrix','ObjectiveSense
wgtAsset = optimvar('asset_weight', p, 1, 'Type', 'continuous', 'LowerBound', 0, 'UpperBound', 1
wgtFactor = optimvar('factor_weight', k, 1, 'Type', 'continuous');

optimProb.Objective = sum(meanStockRetn'.*wgtAsset);

optimProb.Constraints.asset_factor_weight = factorLoading'*wgtAsset - wgtFactor == 0;
optimProb.Constraints.risk = wgtFactor'*covarFactor*wgtFactor + wgtAsset'*D*wgtAsset <= tRisk;
optimProb.Constraints.budget = sum(wgtAsset) == 1;

x0.asset_weight = ones(p, 1)/p;
x0.factor_weight = zeros(k, 1);
opt = optimoptions("fmincon", "Algorithm","sqp", "Display", "off", ...
    'ConstraintTolerance', 1.0e-8, 'OptimalityTolerance', 1.0e-8, 'StepTolerance', 1.0e-8);
x = solve(optimProb,x0, "Options",opt);
assetWgt1 = x.asset_weight;
```

In this example, you are maximizing the portfolio return for a target risk. This is a nonlinear programming problem with a quadratic constraint and you use `fmincon` to solve this problem.

Check for asset allocations that are over 5% to determine which assets have large investment weights.

```
percentage = 0.05;
AssetName = stockReturns.Properties.VariableNames(assetWgt1>=percentage)';
Weight = assetWgt1(assetWgt1>=percentage);
T1 = table(AssetName, Weight)
```

```
T1=7×2 table
     AssetName      Weight 
    ___________    ________

    {'Asset9' }    0.080054
    {'Asset32'}     0.22355
    {'Asset47'}     0.11369
    {'Asset57'}    0.088321
    {'Asset61'}    0.068846
    {'Asset75'}    0.063648
    {'Asset94'}     0.22163
```

Optimize Asset Allocation Using `Portfolio` Class with Factor Information

If you already have the factor loading and factor covariance matrix from some other analysis or third-party provider, you can use this information to compute the asset covariance matrix and then directly run a mean-variance optimization using the `Portfolio` class. Recall that portfolio risk is $\mathrm{Var}\left(R_p\right) = \mathrm{Var}\left(\left(\mu_a + F\, r_f + \varepsilon_a\right)^T w_a\right) = w_a^T\left(F\Sigma_f F^T + D\right) w_a$, so you can obtain the covariance of the asset returns by $\Sigma_a = F\Sigma_f F^T + D$.

Use `estimateFrontierByRisk` in the `Portfolio` class to solve the optimization problem: $\max \mu_a^T\, w_a\,,\ s.t.\ w_a^T\ \Sigma_a\, w_a \le t_{\mathrm{risk}}, 0 \le w_a \le 1,\ e^T w_a = 1$. The `Portfolio` class supports has a variety of built-in constraints that you can use to describe your portfolio problem and estimate the risk and returns on the efficient frontier. For more information, see “Portfolio Set for Optimization Using Portfolio Objects” on page 4-8.

```
covarAsset = factorLoading*covarFactor*factorLoading'+D;
port = Portfolio("AssetMean", meanStockRetn, 'AssetCovar', covarAsset, 'LowerBound', 0, 'UpperBou
    'Budget', 1);
assetWgt2 = estimateFrontierByRisk(port, targetRisk);

AssetName = stockReturns.Properties.VariableNames(assetWgt2>=percentage)';
Weight = assetWgt2(assetWgt2>=percentage);
T2 = table(AssetName, Weight)
```

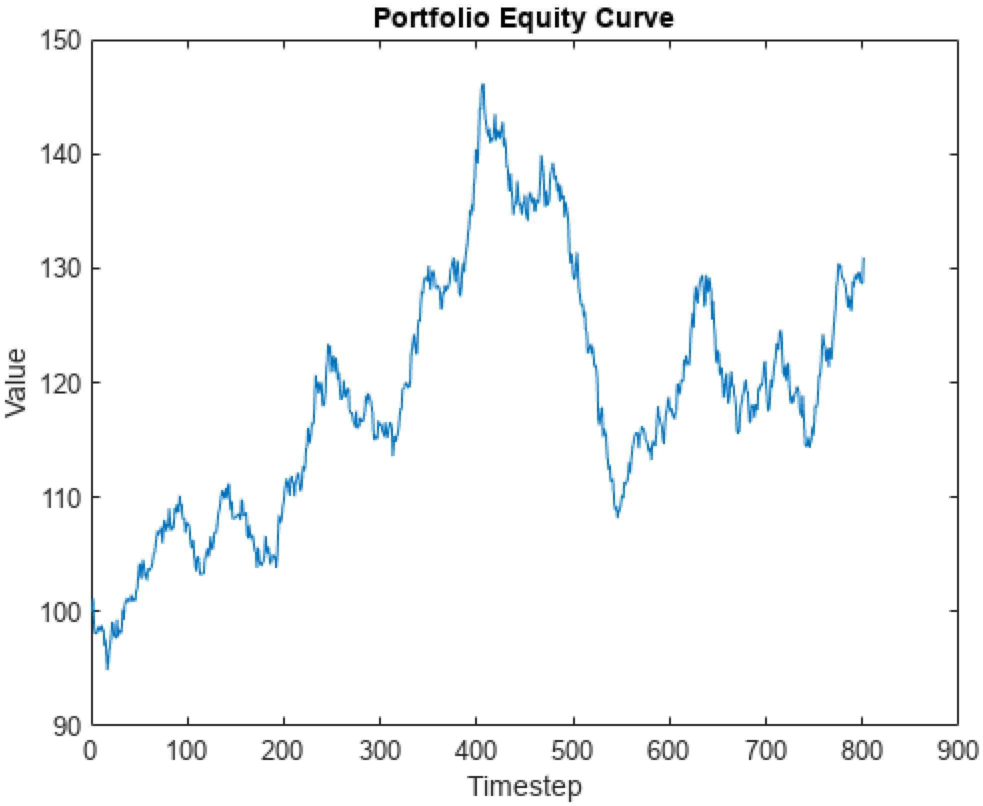
```
T2=7×2 table
     AssetName      Weight 
    ___________    ________

    {'Asset9' }    0.080061
    {'Asset32'}     0.22355
    {'Asset47'}     0.11369
    {'Asset57'}    0.088314
    {'Asset61'}    0.068847
    {'Asset75'}    0.063644
    {'Asset94'}     0.22163
```

Portfolio Optimization Results

Tables `T1` and `T2` show an identical allocation for the asset allocations that are over 5%. Therefore, in this example, both approaches to portfolio optimization with a factor model obtain asset weights that are identical.

Visualize the performance of the optimized allocation over the testing period.

```
retn = stockReturns{test, :}*assetWgt1;
plot(ret2tick(retn, 'method', 'simple')*100); hold off;
xlabel('Timestep');
ylabel('Value');
title('Portfolio Equity Curve');
```



This example demonstrates how to derive statistical factors from asset returns using PCA and then use these factors to perform a factor-based portfolio optimization. This example also shows how to use these statistical factors with the `Portfolio` class. In practice, you can adapt this example to incorporate some measurable market factors, such as industrial factors or ETF returns from various sectors, to describe the randomness in the market [1 on page 4-229]. You can define custom constraints on the weights of factors or assets with high flexibility using the problem-based definition framework from Optimization Toolbox™. Alternatively, you can work directly with the `Portfolio` class to run a portfolio optimization with various built-in constraints.

Reference

1 Lòpez de Prado, M. *Machine Learning for Asset Managers*. Cambridge University Press, 2020.

2 Meucci, A. "Modeling the Market." *Risk and Asset Allocation.* Berlin:Springer, 2009.

See Also

`Portfolio` | `setBounds` | `addGroups` | `setAssetMoments` | `estimateAssetMoments` | `estimateBounds` | `plotFrontier` | `estimateFrontierLimits` | `estimateFrontierByRisk` | `estimatePortRisk`

Related Examples

- “Creating the Portfolio Object” on page 4-24
- “Working with Portfolio Constraints Using Defaults” on page 4-57
- “Validate the Portfolio Problem for Portfolio Object” on page 4-90
- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-94
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-118
- “Postprocessing Results to Set Up Tradable Portfolios” on page 4-126
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-183
- “Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215
- “Portfolio Optimization Against a Benchmark” on page 4-195
- “Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
- “Portfolio Optimization Using Social Performance Measure” on page 4-257
- “Diversify Portfolios Using Custom Objective” on page 4-329

More About

- “Portfolio Object” on page 4-19
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-17

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Backtest Investment Strategies Using Financial Toolbox™

Perform backtesting of portfolio strategies using a backtesting framework. Backtesting is a useful tool to compare how investment strategies perform over historical or simulated market data. This example develops five different investment strategies and then compares their performance after running over a one-year period of historical stock data. The backtesting framework is implemented in two Financial Toolbox™ classes: `backtestStrategy` and `backtestEngine`.

Load Data

Load one year of adjusted price data for 30 stocks. The backtesting frameworks require adjusted asset prices, meaning prices adjusted for dividends, splits, or other events. The prices must be stored in a MATLAB® `timetable` with each column holding a time series of asset prices for an investable asset.

For this example, use one year of asset price data from the component stocks of the Dow Jones Industrial Average.

```
% Read a table of daily adjusted close prices for 2006 DJIA stocks.
T = readtable('dowPortfolio.xlsx');

% For readability, use only 15 of the 30 DJI component stocks.
assetSymbols = ["AA","CAT","DIS","GM","HPQ","JNJ","MCD","MMM","MO","MRK","MSFT","PFE","PG","T","X

% Prune the table to hold only the dates and selected stocks.
timeColumn = "Dates";
T = T(:,[timeColumn assetSymbols]);

% Convert to the table to a timetable.
pricesTT = table2timetable(T,'RowTimes','Dates');

% View the structure of the prices timetable.
head(pricesTT)
```

```
       Dates        AA       CAT      DIS       GM      HPQ      JNJ      MCD      MMM      MO
    ___________    _____    _____    _____    _____    _____    _____    _____    _____    _____

    03-Jan-2006    28.72    55.86    24.18    17.82    28.35    59.08    32.72    75.93    52.27
    04-Jan-2006    28.89    57.29    23.77     18.3    29.18    59.99    33.01    75.54    52.65
    05-Jan-2006    29.12    57.29    24.19    19.34    28.97    59.74    33.05    74.85    52.52
    06-Jan-2006    29.02    58.43    24.52    19.61     29.8    60.01    33.25    75.47    52.95
    09-Jan-2006    29.37    59.49    24.78    21.12    30.17    60.38    33.88    75.84    53.11
    10-Jan-2006    28.44    59.25    25.09    20.79    30.33    60.49    33.91    75.37    53.04
    11-Jan-2006    28.05    59.28    25.33    20.61    30.88    59.91     34.5    75.22    53.31
    12-Jan-2006    27.68    60.13    25.41    19.76    30.57    59.63    33.96    74.57    53.23
```

```
% View the size of the asset price data set.
numSample = size(pricesTT.Variables, 1);
numAssets = size(pricesTT.Variables, 2);
table(numSample, numAssets)
```

```
ans=1×2 table
    numSample    numAssets
    _________    _________
    251         15
```

Define the Strategies

Investment strategies capture the logic used to make asset allocation decisions while a backtest is running. As the backtest runs, each strategy is periodically given the opportunity to update its portfolio allocation based on the trailing market conditions, which it does by setting a vector of asset weights. The asset weights represent the percentage of available capital invested into each asset, with each element in the weights vector corresponding to the respective column in the asset `pricesTT` timetable. If the sum of the weights vector is `1`, then the portfolio is fully invested.

In this example, there are five backtest strategies. The backtest strategies assign asset weights using the following criteria:

- Equal-weighted

$$\omega\mathrm{EW} = (\omega_1, \omega_2, \ldots, \omega_N), \omega_i = \frac{1}{N}$$

- Maximization of Sharpe ratio

$\omega\mathrm{SR} \ = \ \underset{\omega}{\operatorname{argmax}}\left\{\frac{r'\omega}{\sqrt{\omega' Q\omega}} \,|\, \omega \geq 0, \sum_1^N \omega_i = 1, 0 \leq \omega \leq 0.1\right\}$, where r is a vector of expected returns and Q is the covariance matrix of asset returns.

- Inverse variance

$\omega\mathrm{IV} = \ (\omega_1, \omega_2, \ldots, \omega_N), \omega_i = \frac{\left(\sigma_{\mathrm{ii}}^{-1}\right)}{\sum_{i\,=\,1}^{N} \sigma_{\mathrm{ii}}^{-1}}$, where σ_{ii} are diagonal elements of the asset return covariance matrix.

- Markowitz portfolio optimization (maximizing return and minimizing risk with fixed risk-aversion coefficient)

$R_{\mathrm{Mkwtz}} = \max_{\omega}\left\{r'\omega \ - \lambda\omega' Q\omega \mid \omega \geq 0, \sum_1^N \omega_i = 1, 0 \leq \omega \leq 0.1\right\}$, where λ is the risk-aversion coefficient.

- Robust optimization with uncertainty in expected returns
- The robust portfolio optimization strategy, in contrast to the deterministic Markowitz formulation, takes into consideration the uncertainty expected returns of the assets and their variances and covariances. Instead of modeling unknown values (for example, expected returns) as one point, typically represented by the mean value calculated from the past, unknowns are specified as a set of values that contain the most likely possible realizations, $r = \{r \,|\, r \in S(r_0)\}$.

In this case, the expected return is defined not by the deterministic vector r_0 but by the region $S(r_0)$ around the vector r_0.

Taking this into consideration, there are several ways to reformulate the portfolio optimization problem. One of the most frequently used methods is to formulate the problem as a problem of finding the maximum and minimum:

$$R_{\mathrm{robust}} = \max_{\omega} \min_{r \in S(r_0)}\left\{r'\omega \ - \lambda\omega' Q\omega \mid \omega \geq 0, \sum_1^N \omega_i = 1, 0 \leq \omega \leq 0.1\right\}$$

In this example, the region of uncertainty $S(r_0)$ is specified as an ellipsoid:

$$S(r_0) = \{r \mid (r - r_0)' \Sigma_r^{-1} (r - r_0) \le \kappa^2\}$$

Here, κ - is the uncertainty aversion coefficient that defines how wide the uncertainty region is, and Σ_r is the matrix of estimation errors in expected returns r.

With the addition of the ellipsoid uncertainty to the Markowitz model, the robust optimization problem is reformulated as:

$$R_{\text{robust}} = \max_{\omega} \left\{ r'\omega \ - \lambda\omega' Q\omega - \text{kz} \mid \omega \ge 0,\ z \ge 0,\ \omega'\Sigma_r\omega - z^2 \le 0,\ \sum_1^N \omega_i = 1, 0 \le \omega \le 0.1 \right\}$$

Implement the Strategy Rebalance Functions

The core logic of each strategy is implemented in a rebalance function. A rebalance function is a user-defined MATLAB® function that specifies how a strategy allocates capital in a portfolio. The rebalance function is an input argument to `backtestStrategy`. The rebalance function must implement the following fixed signature:

```
function new_weights = allocationFunctionName(current_weights,
pricesTimetable)
```

This fixed signature is the API that the backtest framework uses when rebalancing a portfolio. As the backtest runs, the backtesting engine calls the rebalance function of each strategy, passing in these inputs:

- `current_weights` — Current portfolio weights before rebalancing
- `pricesTimetable` — MATLAB® timetable object containing the rolling window of asset prices.

The `backtestStrategy` rebalance function uses this information to compute the desired new portfolio weights, which are returned to the backtesting engine in the function output `new_weights`. See the Local Functions on page 4-240 sections for the rebalance function for each of the five strategies.

Compute Initial Strategy Weights

Use the strategy rebalance functions to compute the initial weights for each strategy. Setting the initial weights is important because otherwise the strategies begin the backtest with 100% in cash, earning the risk-free rate, until the first rebalance date.

This example uses the first 40 days of the data set (about 2 months) to initialize the strategies. The backtest is then run over the remaining data (about 10 months).

```
warmupPeriod = 40;
```

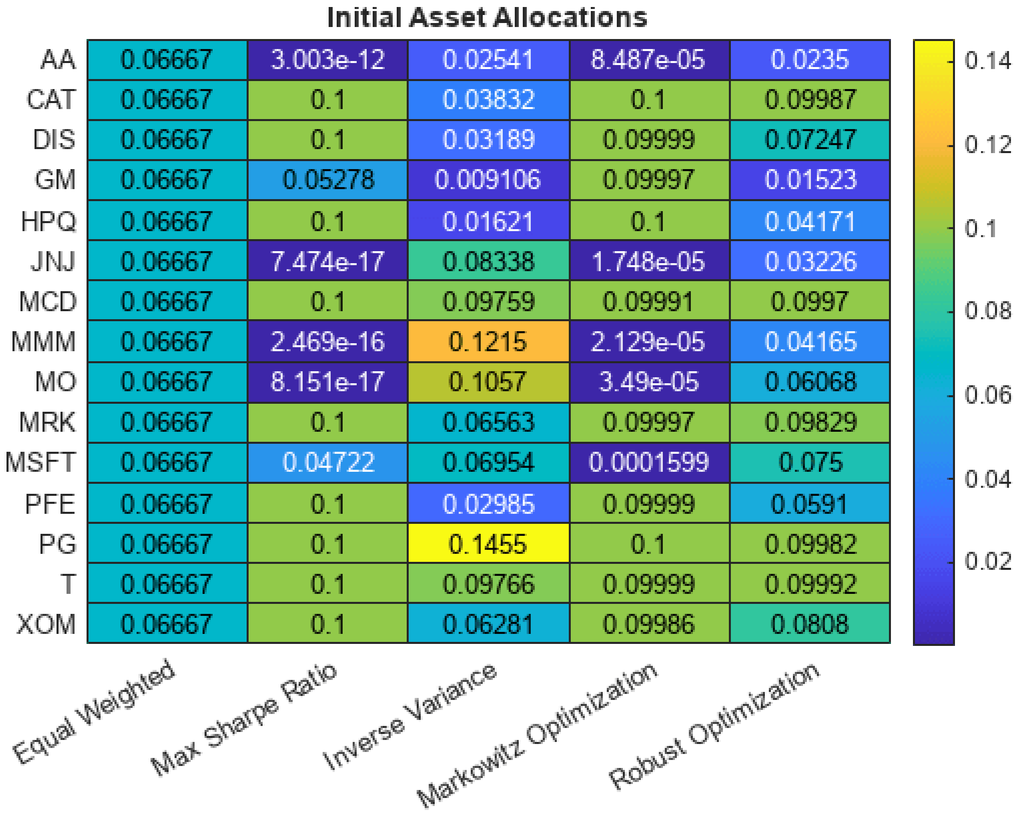
The initial weights are calculated by calling the `backtestStrategy` rebalance function in the same way that the backtesting engine will call it. To do so, pass in a vector of current weights (all zeros, that is 100% cash) as well as a window of price data that the strategies will use to set the desired weights (the warm-up data partition). Using the rebalance functions to compute the initial weights in this way is not required. The initial weights are a vector of the initial portfolio weights and can be set to any appropriate value. The rebalance functions in this example approximate the state the strategies would be in had they been already running at the start of the backtest.

```
% No current weights (100% cash position).
current_weights = zeros(1,numAssets);
```

```
% Warm-up partition of data set timetable.
warmupTT = pricesTT(1:warmupPeriod,:);

% Compute the initial portfolio weights for each strategy.
equalWeight_initial     = equalWeightFcn(current_weights,warmupTT);
maxSharpeRatio_initial  = maxSharpeRatioFcn(current_weights,warmupTT);
inverseVariance_initial = inverseVarianceFcn(current_weights,warmupTT);
markowitz_initial       = markowitzFcn(current_weights,warmupTT);
robustOptim_initial     = robustOptimFcn(current_weights,warmupTT);
```

Visualize the initial weight allocations from the strategies.

```
strategyNames = {'Equal Weighted', 'Max Sharpe Ratio', 'Inverse Variance', 'Markowitz Optimizati
assetSymbols = pricesTT.Properties.VariableNames;
initialWeights = [equalWeight_initial(:), maxSharpeRatio_initial(:), inverseVariance_initial(:),
heatmap(strategyNames, assetSymbols, initialWeights, 'title','Initial Asset Allocations','Colorma
```



Create Backtest Strategies

To use the strategies in the backtesting framework, you must build `backtestStrategy` objects, one for each strategy. The `backtestStrategy` function takes as input the strategy name and rebalancing function for each strategy. Additionally, the `backtestStrategy` can take a variety of name-value pair arguments to specify various options. For more information on creating backtest strategies, see `backtestStrategy`.

Set the rebalance frequency and lookback window size are set in terms of number of time steps (that is, rows of the `pricesTT` timetable). Since the data is daily price data, specify the rebalance frequency and rolling lookback window in days.

```
% Rebalance approximately every 1 month (252 / 12 = 21).
rebalFreq = 21;

% Set the rolling lookback window to be at least 40 days and at most 126
% days (about 6 months).
lookback  = [40 126];

% Use a fixed transaction cost (buy and sell costs are both 0.5% of amount
% traded).
transactionsFixed = 0.005;

% Customize the transaction costs using a function. See the
% variableTransactionCosts function below for an example.
transactionsVariable = @variableTransactionCosts;

% The first two strategies use fixed transaction costs. The equal-weighted
% strategy does not require a lookback window of trailing data, as its
% allocation is fixed.
strat1 = backtestStrategy('Equal Weighted', @equalWeightFcn, ...
    'RebalanceFrequency', rebalFreq, ...
    'LookbackWindow', 0, ...
    'TransactionCosts', transactionsFixed, ...
    'InitialWeights', equalWeight_initial);

strat2 = backtestStrategy('Max Sharpe Ratio', @maxSharpeRatioFcn, ...
    'RebalanceFrequency', rebalFreq, ...
    'LookbackWindow', lookback, ...
    'TransactionCosts', transactionsFixed, ...
    'InitialWeights', maxSharpeRatio_initial);

% Use variable transaction costs for the remaining strategies.
strat3 = backtestStrategy('Inverse Variance', @inverseVarianceFcn, ...
    'RebalanceFrequency', rebalFreq, ...
    'LookbackWindow', lookback, ...
    'TransactionCosts', @variableTransactionCosts, ...
    'InitialWeights', inverseVariance_initial);

strat4 = backtestStrategy('Markowitz Optimization', @markowitzFcn, ...
    'RebalanceFrequency', rebalFreq, ...
    'LookbackWindow', lookback, ...
    'TransactionCosts', transactionsFixed, ...
    'InitialWeights', markowitz_initial);

strat5 = backtestStrategy('Robust Optimization', @robustOptimFcn, ...
    'RebalanceFrequency', rebalFreq, ...
    'LookbackWindow', lookback, ...
    'TransactionCosts', transactionsFixed, ...
    'InitialWeights', robustOptim_initial);

% Aggregate the strategy objects into an array.
strategies = [strat1, strat2, strat3, strat4, strat5];
```

Backtest the Strategies

Use the following the workflow to backtest the strategies with a `backtestEngine`.

Define Backtesting Engine

The `backtestEngine` function takes as input an array of `backtestStrategy` objects. Additionally, when using `backtestEngine`, you can set several options, such as the risk-free rate and the initial portfolio value. When the risk-free rate is specified in annualized terms, the `backtestEngine` uses `Basis` property to set the day count convention. For more information on creating backtesting engines, see `backtestEngine`.

```
% Risk-free rate is 1% annualized
annualRiskFreeRate = 0.01;

% Create the backtesting engine object
backtester = backtestEngine(strategies, 'RiskFreeRate', annualRiskFreeRate)
```

```
backtester = 
  backtestEngine with properties:

               Strategies: [1x5 backtestStrategy]
             RiskFreeRate: 0.0100
           CashBorrowRate: 0
          RatesConvention: "Annualized"
                    Basis: 0
    InitialPortfolioValue: 10000
           DateAdjustment: "Previous"
                NumAssets: []
                  Returns: []
                Positions: []
                 Turnover: []
                  BuyCost: []
                 SellCost: []
                     Fees: []
```

Run Backtest

Use `runBacktest` to run the backtest using the test data partition. Use the `runBacktest` name-value pair argument `'Start'`to avoid look-ahead bias (that is, "seeing the future"). Begin the backtest at the end of the "warm-up" period. Running the backtest populates the empty fields of the `backtestEngine` object with the day-by-day backtest results.

```
backtester = runBacktest(backtester, pricesTT, 'Start', warmupPeriod)
```

```
backtester = 
  backtestEngine with properties:

               Strategies: [1x5 backtestStrategy]
             RiskFreeRate: 0.0100
           CashBorrowRate: 0
          RatesConvention: "Annualized"
                    Basis: 0
    InitialPortfolioValue: 10000
           DateAdjustment: "Previous"
                NumAssets: 15
                  Returns: [211x5 timetable]
                Positions: [1x1 struct]
                 Turnover: [211x5 timetable]
                  BuyCost: [211x5 timetable]
                 SellCost: [211x5 timetable]
                     Fees: [1x1 struct]
```

Examine Backtest Results

Use the `summary` function to generate a table of strategy performance results for the backtest.

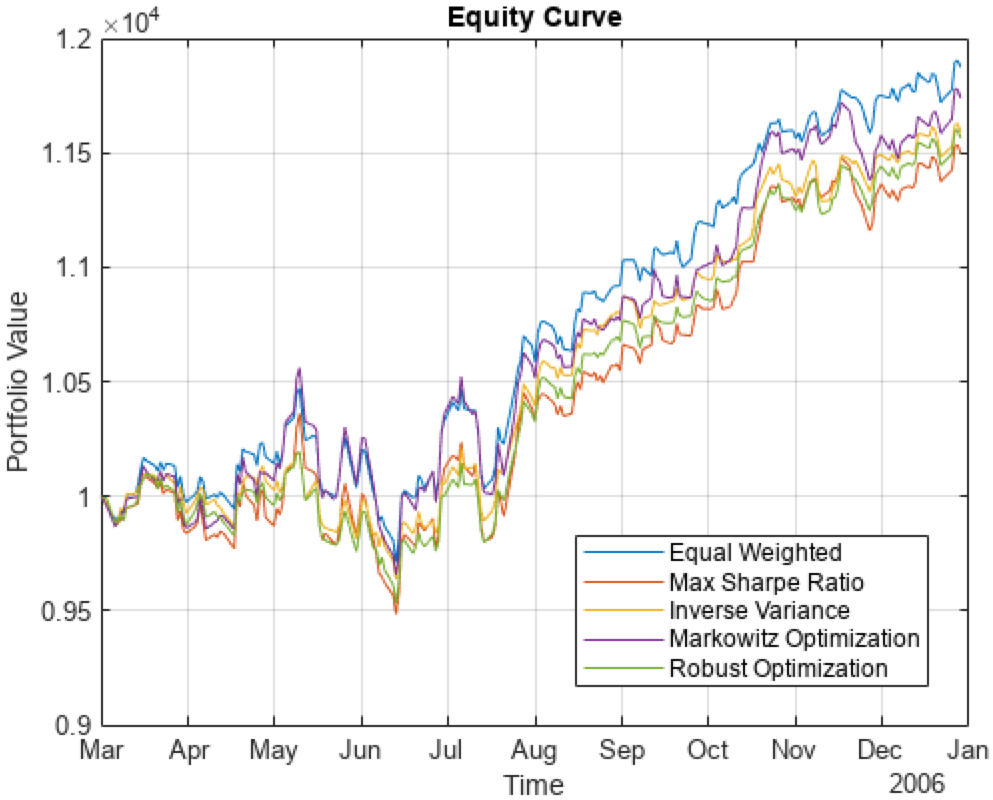
```
summaryByStrategies = summary(backtester)
```

```
summaryByStrategies=9×5 table
                       Equal_Weighted    Max_Sharpe_Ratio    Inverse_Variance    Markowitz_Optimi
                       ______________    ________________    ________________    ________________

    TotalReturn             0.18745             0.14991             0.15906                 0.17404
    SharpeRatio             0.12559            0.092456             0.12179                 0.10339
    Volatility            0.0063474           0.0070186           0.0055626               0.0072466
    AverageTurnover      0.00087623           0.0065762           0.0028666               0.0058268
    MaxTurnover            0.031251               0.239             0.09114                 0.21873
    AverageReturn        0.00083462          0.00068672           0.0007152              0.00078682
    MaxDrawdown            0.072392            0.084768            0.054344                0.085544
    AverageBuyCost         0.047298              0.3449             0.15228                  0.3155
    AverageSellCost        0.047298              0.3449             0.22842                  0.3155
```

The detailed backtest results, including the daily returns, asset positions, turnover, and fees are stored in properties of the `backtestEngine` object.

Use `equityCurve` to plot the equity curve for the five different investment strategies.

```
equityCurve(backtester)
```

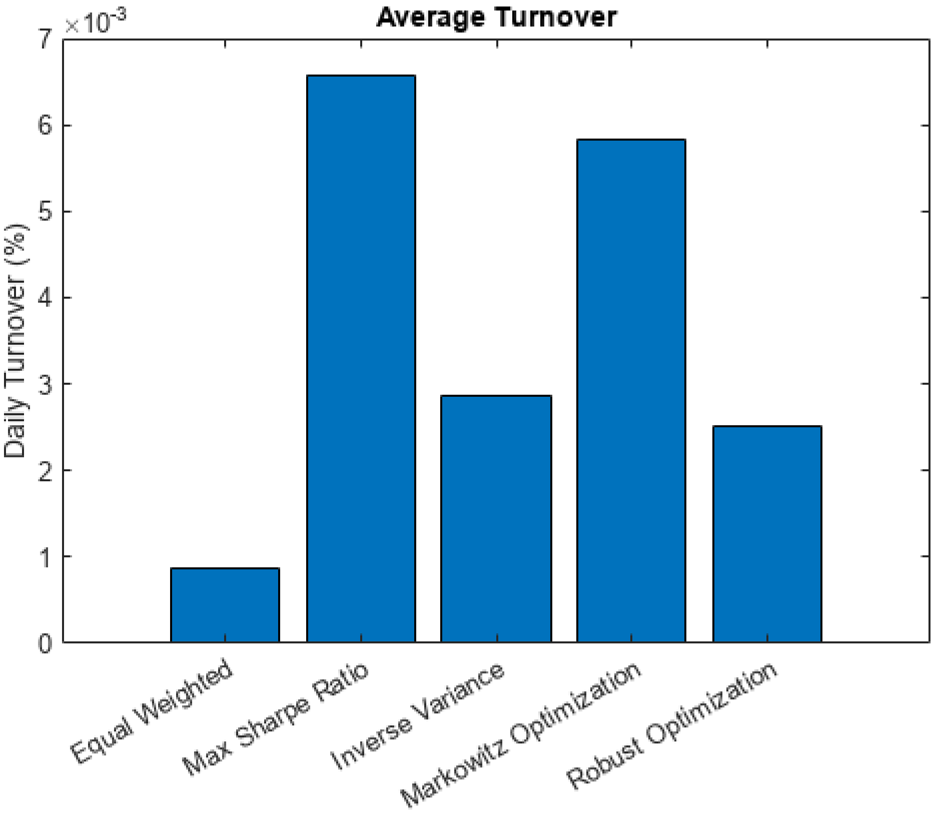


Transposing the summary table to make plots of certain metrics can be useful.

```
% Transpose the summary table to plot the metrics.
summaryByMetrics = rows2vars(summaryByStrategies);
summaryByMetrics.Properties.VariableNames{1} = 'Strategy'
```

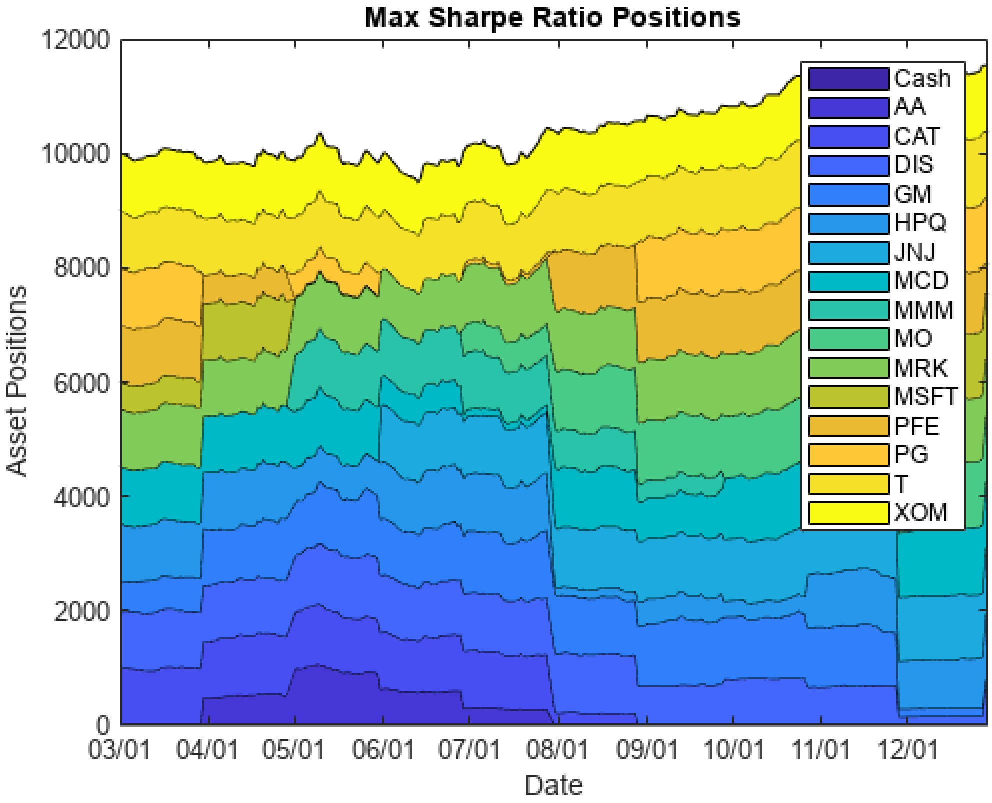
```
summaryByMetrics=5×10 table
             Strategy                 TotalReturn    SharpeRatio    Volatility    AverageTurnover
    __________________________        ___________    ___________    __________    _______________

    {'Equal_Weighted'         }         0.18745        0.12559      0.0063474       0.00087623
    {'Max_Sharpe_Ratio'       }         0.14991       0.092456      0.0070186        0.0065762
    {'Inverse_Variance'       }         0.15906        0.12179      0.0055626        0.0028666
    {'Markowitz_Optimization'}          0.17404        0.10339      0.0072466        0.0058268
    {'Robust_Optimization'    }         0.15655        0.11442      0.0058447        0.0025172
```

```
% Compare the strategy turnover.
names = [backtester.Strategies.Name];
nameLabels = strrep(names,'_',' ');
bar(summaryByMetrics.AverageTurnover)
title('Average Turnover')
ylabel('Daily Turnover (%)')
set(gca,'xticklabel',nameLabels)
```



You can visualize the change in the strategy allocations over time using an area chart of the daily asset positions. For information on the `assetAreaPlot` function, see the Local Functions on page 4-243 section.

```
strategyName = Max Sharpe Ratio ▼;
assetAreaPlot(backtester,strategyName)
```



Local Functions

The strategy rebalancing functions as well as the variable transaction cost function follow.

```
function new_weights = equalWeightFcn(current_weights, pricesTT)
% Equal-weighted portfolio allocation

nAssets = size(pricesTT, 2);
new_weights = ones(1,nAssets);
new_weights = new_weights / sum(new_weights);

end

function new_weights = maxSharpeRatioFcn(current_weights, pricesTT)
% Mean-variance portfolio allocation

nAssets = size(pricesTT, 2);
assetReturns = tick2ret(pricesTT);
% Max 25% into a single asset (including cash)
p = Portfolio('NumAssets',nAssets,...
    'LowerBound',0,'UpperBound',0.1,...
    'LowerBudget',1,'UpperBudget',1);
p = estimateAssetMoments(p, assetReturns{:,:});
new_weights = estimateMaxSharpeRatio(p);

end
```

```
function new_weights = inverseVarianceFcn(current_weights, pricesTT) 
% Inverse-variance portfolio allocation

assetReturns = tick2ret(pricesTT);
assetCov = cov(assetReturns{:,:});
new_weights = 1 ./ diag(assetCov);
new_weights = new_weights / sum(new_weights);

end

function new_weights = robustOptimFcn(current_weights, pricesTT) 
% Robust portfolio allocation

nAssets = size(pricesTT, 2);
assetReturns = tick2ret(pricesTT);

Q = cov(table2array(assetReturns));
SIGMAx = diag(diag(Q));

% Robust aversion coefficient
k = 1.1;

% Robust aversion coefficient
lambda = 0.05;

rPortfolio = mean(table2array(assetReturns))';

% Create the optimization problem
pRobust = optimproblem('Description','Robust Portfolio');

% Define the variables
% xRobust - x  allocation vector
xRobust = optimvar('x',nAssets,1,'Type','continuous','LowerBound',0.0,'UpperBound',0.1);
zRobust = optimvar('z','LowerBound',0);

% Define the budget constraint
pRobust.Constraints.budget = sum(xRobust) == 1;

% Define the robust constraint
pRobust.Constraints.robust = xRobust'*SIGMAx*xRobust - zRobust*zRobust <=0;
pRobust.Objective = -rPortfolio'*xRobust + k*zRobust + lambda*xRobust'*Q*xRobust;
x0.x = zeros(nAssets,1);
x0.z = 0;
opt = optimoptions('fmincon','Display','off');
[solRobust,~,~] = solve(pRobust,x0,'Options',opt);
new_weights = solRobust.x;

end

function new_weights = markowitzFcn(current_weights, pricesTT) 
% Robust portfolio allocation

nAssets = size(pricesTT, 2);
assetReturns = tick2ret(pricesTT);

Q = cov(table2array(assetReturns));

% Risk aversion coefficient
```

```
lambda = 0.05;

rPortfolio = mean(table2array(assetReturns))';

% Create the optimization problem
pMrkwtz = optimproblem('Description','Markowitz Mean Variance Portfolio ');

% Define the variables
% xRobust - x  allocation vector
xMrkwtz = optimvar('x',nAssets,1,'Type','continuous','LowerBound',0.0,'UpperBound',0.1);

% Define the budget constraint
pMrkwtz.Constraints.budget = sum(xMrkwtz) == 1;

% Define the Markowitz objective
pMrkwtz.Objective = -rPortfolio'*xMrkwtz + lambda*xMrkwtz'*Q*xMrkwtz;
x0.x = zeros(nAssets,1);

opt = optimoptions('quadprog','Display','off');
[solMrkwtz,~,~] = solve(pMrkwtz,x0,'Options',opt);
new_weights = solMrkwtz.x;

end

function [buy, sell] = variableTransactionCosts(deltaPositions)
% Variable transaction cost function
%
% This function is an example of how to compute variable transaction costs.
%
% Compute scaled transaction costs based on the change in market value of
% each asset after a rebalance.  Costs are computed at the following rates:
%
% Buys:
%   $0-$10,000 : 0.5%
%   $10,000+   : 0.35%
% Sells:
%   $0-$1,000  : 0.75%
%   $1,000+    : 0.5%

buy  = zeros(1,numel(deltaPositions));
sell = zeros(1,numel(deltaPositions));

% Buys
idx = 0 < deltaPositions & deltaPositions < 1e4;
buy(idx) = 0.005 * deltaPositions(idx); % 50 basis points
idx = 1e4 <= deltaPositions;
buy(idx) = 0.0035 * deltaPositions(idx); % 35 basis ponits
buy = sum(buy);

% Sells
idx = -1e3 < deltaPositions & deltaPositions < 0;
sell(idx) = 0.0075 * -deltaPositions(idx); % 75 basis points
idx = deltaPositions <= -1e3;
sell(idx) = 0.005 * -deltaPositions(idx); % 50 basis points
sell = sum(sell);

end
```

```
function assetAreaPlot(backtester,strategyName)
% Plot the asset allocation as an area plot.

t = backtester.Positions.(strategyName).Time;
positions = backtester.Positions.(strategyName).Variables;
h = area(t,positions);
title(sprintf('%s Positions',strrep(strategyName,'_',' ')));
xlabel('Date');
ylabel('Asset Positions');
datetick('x','mm/dd','keepticks');
xlim([t(1) t(end)])
oldylim = ylim;
ylim([0 oldylim(2)]);
cm = parula(numel(h));
for i = 1:numel(h)
    set(h(i),'FaceColor',cm(i,:));
end
legend(backtester.Positions.(strategyName).Properties.VariableNames)

end
```

See Also

backtestStrategy | backtestEngine | runBacktest | summary

Related Examples

- “Backtest Investment Strategies with Trading Signals” on page 4-244
- “Backtest Strategies Using Deep Learning” on page 4-298
- “Backtest with Brinson Attribution to Evaluate Portfolio Performance” on page 4-311

Backtest Investment Strategies with Trading Signals

This example shows how to perform backtesting of portfolio strategies that incorporate investment signals in their trading strategy. The term *signals* includes any information that a strategy author needs to make with respect to trading decisions outside of the price history of the assets. Such information can include technical indicators, the outputs of machine learning models, sentiment data, macroeconomic data, and so on. This example uses three simple investment strategies based on derivative signal data:

- Moving average crossovers
- Moving average convergence/divergence
- Relative strength index

In this example you can run a backtest using these strategies over one year of stock data. You then analyze the results to compare the performance of each strategy.

Even though technical indicators are not typically used as standalone trading strategies, this example uses these strategies to demonstrate how to build investment strategies based on signal data when you use the `backtestEngine` object in MATLAB®.

Load Data

Load the adjusted price data for 15 stocks for the year 2006. This example uses a small set of investable assets for readability.

Read a table of daily adjusted close prices for 2006 DJIA stocks.

```
T = readtable('dowPortfolio.xlsx');
```

For readability, use only 15 of the 30 DJI component stocks.

```
symbols = ["AA","CAT","DIS","GM","HPQ","JNJ","MCD","MMM","MO","MRK","MSFT","PFE","PG","T","XOM"]
```

Prune the table to hold only the dates and selected stocks.

```
timeColumn = "Dates";
T = T(:,[timeColumn symbols]);
```

Convert the data to a timetable.

```
pricesTT = table2timetable(T,'RowTimes','Dates');
```

View the structure of the prices timetable.

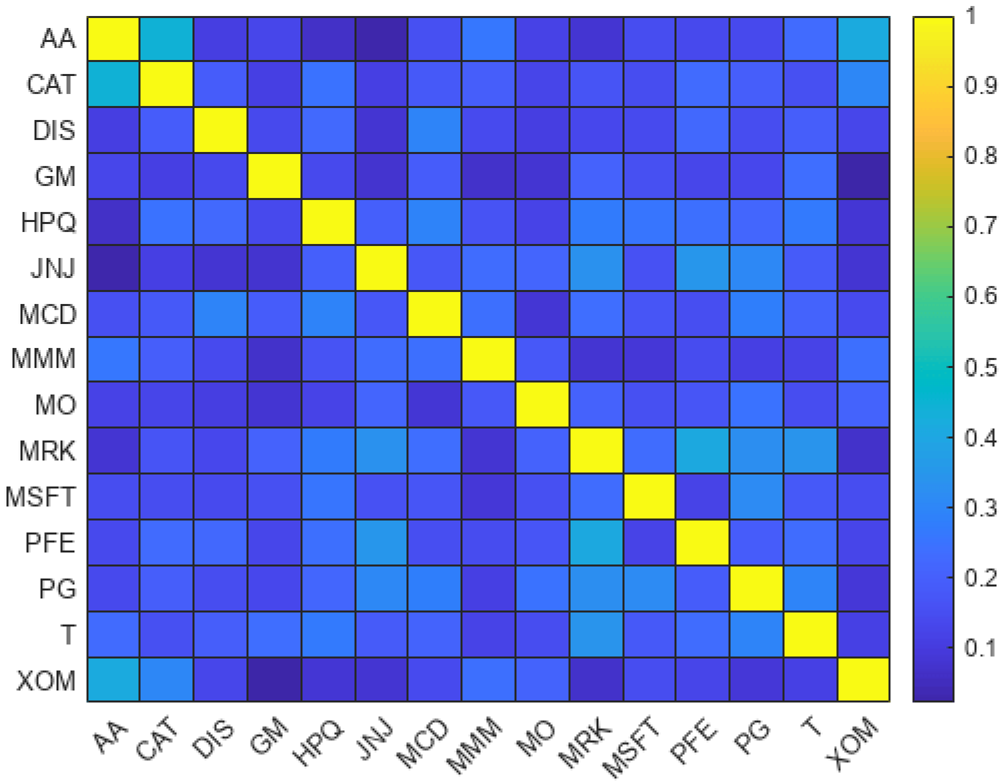
```
head(pricesTT)
```

```
       Dates         AA       CAT      DIS      GM       HPQ      JNJ      MCD      MMM      MO
    ___________    _____    _____    _____    _____    _____    _____    _____    _____    _____

    03-Jan-2006    28.72    55.86    24.18    17.82    28.35    59.08    32.72    75.93    52.27
    04-Jan-2006    28.89    57.29    23.77     18.3    29.18    59.99    33.01    75.54    52.65
    05-Jan-2006    29.12    57.29    24.19    19.34    28.97    59.74    33.05    74.85    52.52
    06-Jan-2006    29.02    58.43    24.52    19.61     29.8    60.01    33.25    75.47    52.95
    09-Jan-2006    29.37    59.49    24.78    21.12    30.17    60.38    33.88    75.84    53.11
    10-Jan-2006    28.44    59.25    25.09    20.79    30.33    60.49    33.91    75.37    53.04
    11-Jan-2006    28.05    59.28    25.33    20.61    30.88    59.91     34.5    75.22    53.31
    12-Jan-2006    27.68    60.13    25.41    19.76    30.57    59.63    33.96    74.57    53.23
```

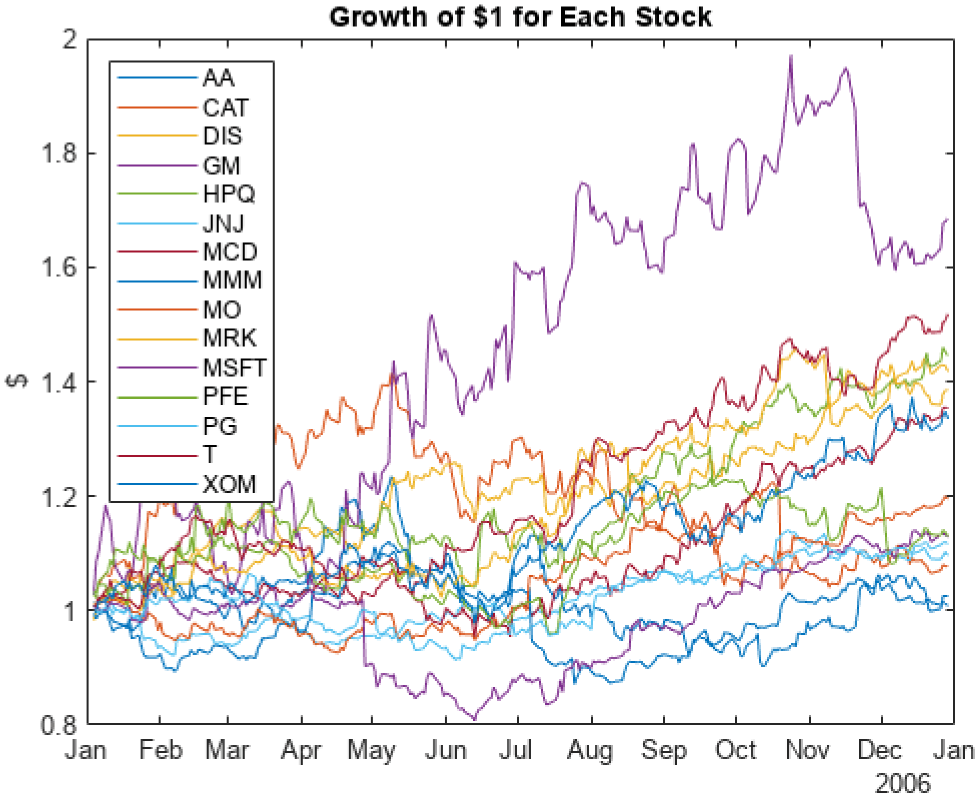
Inspect Data Set

Visualize the correlation and total return of each stock in the data set.

```
% Visualize the correlation between the 15 stocks.
returns = tick2ret(pricesTT);
stockCorr = corr(returns.Variables);
heatmap(symbols,symbols,stockCorr,'Colormap',parula);
```



```
% Visualize the performance of each stock over the range of price data.
totalRet = ret2tick(returns);
plot(totalRet.Dates,totalRet.Variables);
legend(symbols,'Location','NW');
title('Growth of $1 for Each Stock')
ylabel('$')
```



```
% Get the total return of each stock for the duration of the data set.
totalRet(end,:)
```

```
ans=1×15 timetable
       Dates          AA       CAT       DIS       GM       HPQ       JNJ       MCD       MMM
    ___________     ______    ______    ______    ______    ______    ______    ______    ______

    29-Dec-2006     1.0254    1.0781    1.4173    1.6852    1.4451    1.0965    1.3548    1.0087
```

Build Signal Table

In addition to the historical adjusted asset prices, the backtesting framework allows you to optionally specify *signal* data when running a backtest. Specify the signal data in a similar way as the prices by using a MATLAB® `timetable`. The "time" dimension of the *signal* timetable must match that of the *prices* timetable — that is, the rows of each table must have matching datetime values for the `Time` column.

This example builds a signal timetable to support each of the three investment strategies:

- Simple moving average crossover (SMA) strategy
- Moving Average Convergence / Divergence (MACD) strategy
- Relative Strength Index (RSI) strategy

Each strategy has a timetable of signals that are precomputed. Before you run the backtest, you merge the three separate signal timetables into a single aggregate signal timetable to use for the backtest.

SMA: Simple Moving Average Crossover

The SMA indicator uses 5-day and 20-day simple moving averages to make buy and sell decisions. When the 5-day SMA crosses the 20-day SMA (moving upwards), then the stock is bought. When the 5-day SMA crosses below the 20-day SMA, the stock is sold.

```
% Create SMA timetables using the movavg function.
sma5  = movavg(pricesTT,'simple',5);
sma20 = movavg(pricesTT,'simple',20);
```

Create the SMA indicator signal timetable.

```
smaSignalNameEnding = '_SMA5over20';

smaSignal = timetable;
for i = 1:numel(symbols)
    symi = symbols(i);
    % Build a timetable for each symbol, then aggregate them together.
    smaSignali = timetable(pricesTT.Dates,...
        double(sma5.(symi) > sma20.(symi)),...
        'VariableNames',{sprintf('%s%s',symi,smaSignalNameEnding)});
    % Use the synchronize function to merge the timetables together.
    smaSignal = synchronize(smaSignal,smaSignali);
end
```

The SMA signal timetable contains an indicator with a value of `1` when the 5-day moving average is above the 20-day moving average for each asset, and a `0` otherwise. The column names for each stock indicator are [*stock symbol*]`SMA5over20`. The `backtestStrategy` object makes trading decisions based on these crossover events.

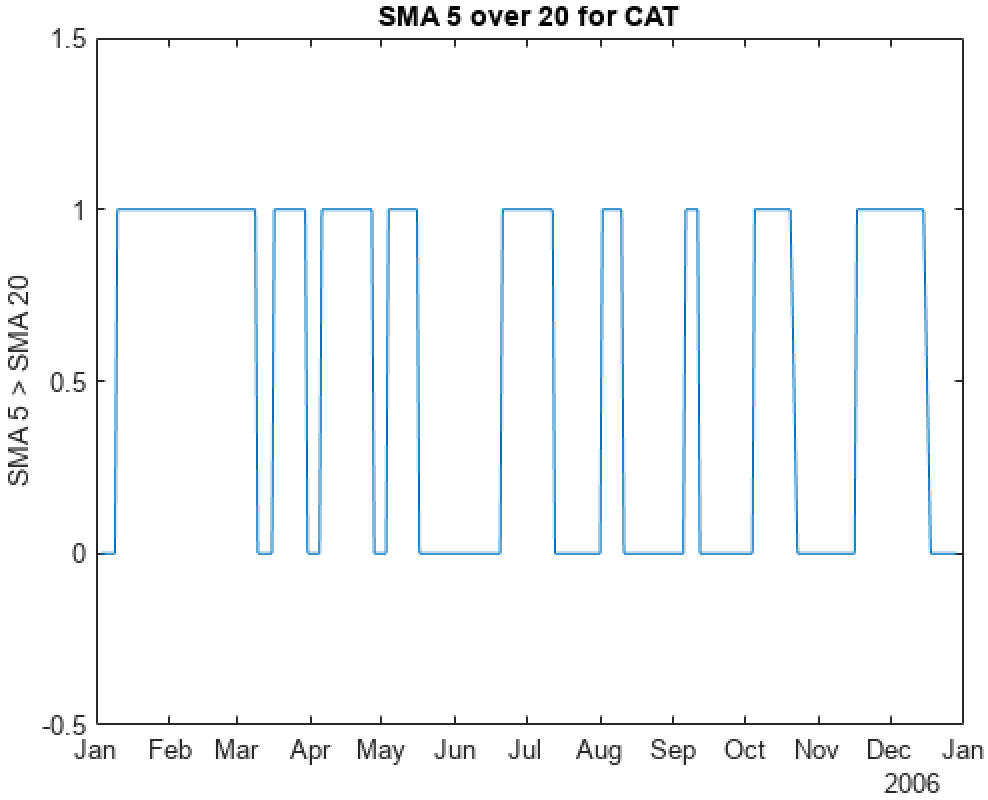
View the structure of the SMA signal timetable.

```
head(smaSignal)
```

```
       Time        AA_SMA5over20    CAT_SMA5over20    DIS_SMA5over20    GM_SMA5over20    HPQ_SMA5
    ___________    _____________    ______________    ______________    _____________    ________

    03-Jan-2006          0                0                 0                 0                0
    04-Jan-2006          0                0                 0                 0                0
    05-Jan-2006          0                0                 0                 0                0
    06-Jan-2006          0                0                 0                 0                0
    09-Jan-2006          0                0                 0                 0                0
    10-Jan-2006          1                1                 1                 1                1
    11-Jan-2006          0                1                 1                 1                1
    12-Jan-2006          0                1                 1                 1                1
```

Plot the signal for a single asset to preview the trading frequency.

```
plot(smaSignal.Time,smaSignal.CAT_SMA5over20);
ylim([-0.5, 1.5]);
ylabel('SMA 5 > SMA 20');
title(sprintf('SMA 5 over 20 for CAT'));
```



MACD: Moving Average Convergence/Divergence

You can use the MACD metric in a variety of ways. Often, MACD is compared to its own exponential moving average, but for this example, MACD serves as a trigger for a buy signal when the MACD rises above 0. A position is sold when the MACD falls back below 0.

```
% Create a timetable of the MACD metric using the MACD function.
macdTT = macd(pricesTT);
```

Create the MACD indicator signal timetable.

```
macdSignalNameEnding = '_MACD';

macdSignal = timetable;
for i = 1:numel(symbols)
    symi = symbols(i);
    % Build a timetable for each symbol, then aggregate the symbols together.
    macdSignali = timetable(pricesTT.Dates,...
        double(macdTT.(symi) > 0),...
        'VariableNames',{sprintf('%s%s',symi,macdSignalNameEnding)});
    macdSignal = synchronize(macdSignal,macdSignali);
end
```

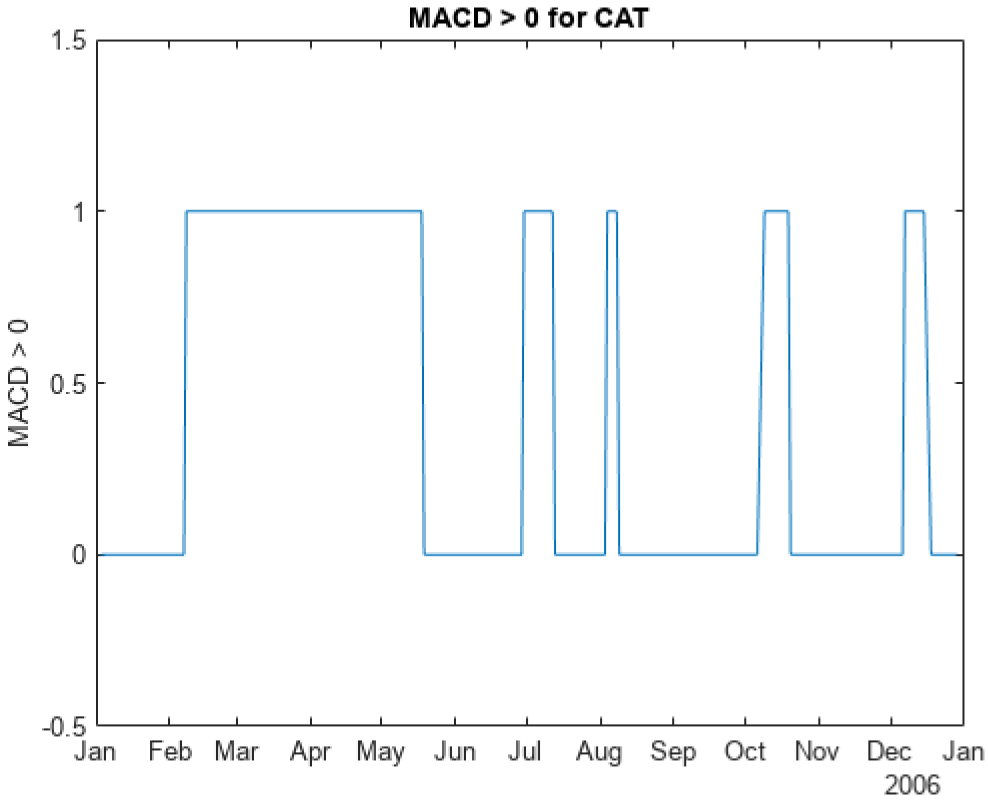
The MACD signal table contains a column for each asset with the name [*stock symbol*]MACD. Each signal has a value of 1 when the MACD of the stock is above 0. The signal has a value of 0 when the MACD of the stock falls below 0.

```
head(macdSignal)
```

```
       Time        AA_MACD    CAT_MACD    DIS_MACD    GM_MACD    HPQ_MACD    JNJ_MACD    MCD_MACD
    ___________    _______    ________    ________    _______    ________    ________    ________

    03-Jan-2006       0           0           0          0           0           0           0
    04-Jan-2006       0           0           0          0           0           0           0
    05-Jan-2006       0           0           0          0           0           0           0
    06-Jan-2006       0           0           0          0           0           0           0
    09-Jan-2006       0           0           0          0           0           0           0
    10-Jan-2006       0           0           0          0           0           0           0
    11-Jan-2006       0           0           0          0           0           0           0
    12-Jan-2006       0           0           0          0           0           0           0
```

Similar to the SMA, plot the signal for a single asset to preview the trading frequency.

```
plot(macdSignal.Time,macdSignal.CAT_MACD)
ylim([-0.5, 1.5]);
ylabel('MACD > 0');
title(sprintf('MACD > 0 for CAT'));
```



RSI: Relative Strength Index

The RSI is a metric to capture momentum. A common heuristic is to buy when the RSI falls below `30` and to sell when the RSI rises above `70`.

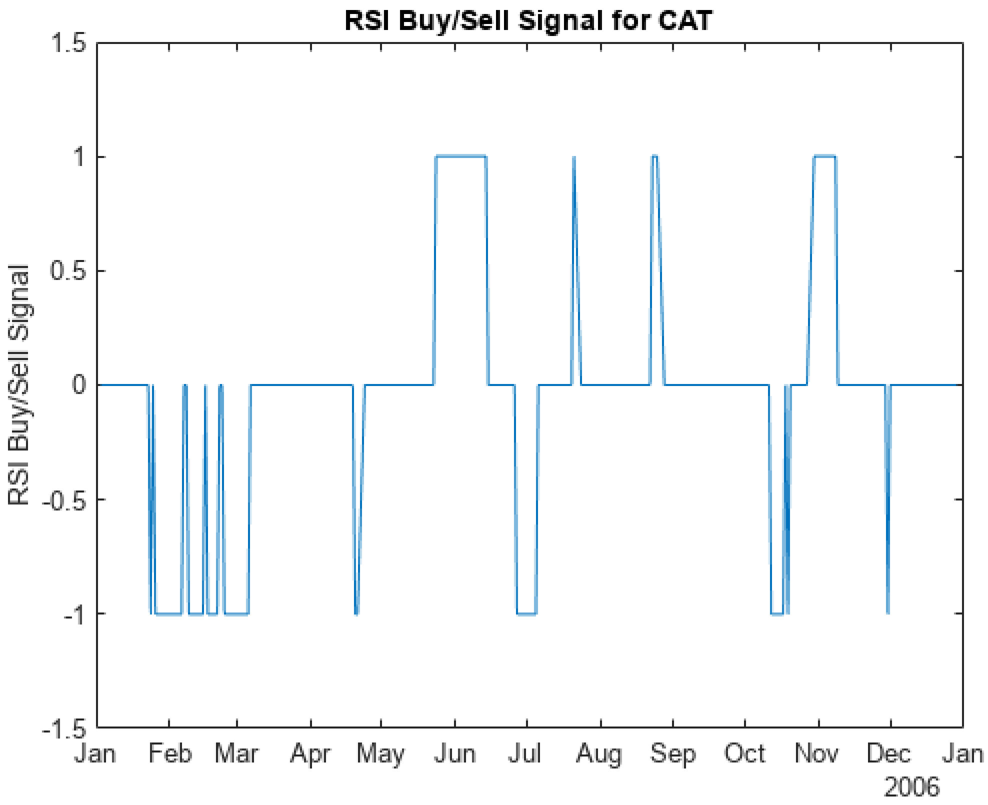
```
rsiSignalNameEnding = '_RSI';

rsiSignal = timetable;
for i = 1:numel(symbols)
```

```
    symi = symbols(i);
    rsiValues = rsindex(pricesTT.(symi));
    rsiBuySell = zeros(size(rsiValues));
    rsiBuySell(rsiValues < 30) = 1;
    rsiBuySell(rsiValues > 70) = -1;
    % Build a timetable for each symbol, then aggregate the symbols together.
    rsiSignali = timetable(pricesTT.Dates,...
        rsiBuySell,...
        'VariableNames',{sprintf('%s%s',symi,rsiSignalNameEnding)});
    rsiSignal = synchronize(rsiSignal,rsiSignali);
end
```

The RSI signal takes a value of 1 (indicating a buy signal) when the RSI value for the stock falls below 30. The signal takes a value of -1 (indicating a sell signal) when the RSI for the stock rises above 70. Otherwise, the signal takes a value of 0, indicating no action.

Plot the signal for a single asset to preview the trading frequency.

```
plot(rsiSignal.Time,rsiSignal.CAT_RSI)
ylim([-1.5, 1.5]);
ylabel('RSI Buy/Sell Signal');
title(sprintf('RSI Buy/Sell Signal for CAT'));
```



Build the Strategies

Build the strategies for the backtestStrategy object using the rebalance functions defined in the Local Functions on page 4-254 section. Each strategy uses the rebalance function to make trading decisions based on the appropriate signals.

The signals require sufficient trailing data to compute the trading signals (for example, computing the SMA20 for day *X* requires prices from the 20 days prior to day *X*). All of the trailing data is captured in the precomputed trading signals. So the actual strategies need only a 2-day lookback window to make trading decisions to evaluate when the signals cross trading thresholds.

All strategies pay 25 basis points transaction costs on buys and sells.

The initial weights are computed based on the signal values after 20 trading days. The backtest begins after this 20 day initialization period.

```
tradingCosts = 0.0025;

% Use the crossoverRebalanceFunction for both the SMA
% strategy as well as the MACD strategy.  This is because they both trade
% on their respective signals in the same way (buy when signal goes from
% 0->1, sell when signal goes from 1->0).  Build an anonymous
% function for the rebalance functions of the strategies that calls the
% shared crossoverRebalanceFcn() with the appropriate signal name string
% for each strategy.

% Each anonymous function takes the current weights (w), prices (p),
% and signal (s) data from the backtest engine and passes it to the
% crossoverRebalanceFcn function with the signal name string.
smaInitWeights = computeInitialWeights(smaSignal(20,:));
smaRebalanceFcn = @(w,p,s) crossoverRebalanceFcn(w,p,s,smaSignalNameEnding);
smaStrategy = backtestStrategy('SMA',smaRebalanceFcn,...
    'TransactionCosts',tradingCosts,...
    'LookbackWindow',2,...
    'InitialWeights',smaInitWeights);

macdInitWeights = computeInitialWeights(macdSignal(20,:));
macdRebalanceFcn = @(w,p,s) crossoverRebalanceFcn(w,p,s,macdSignalNameEnding);
macdStrategy = backtestStrategy('MACD',macdRebalanceFcn,...
    'TransactionCosts',tradingCosts,...
    'LookbackWindow',2,...
    'InitialWeights',macdInitWeights);

% The RSI strategy uses its signal differently, buying on a 0->1
% transition and selling on a 0->-1 transition.  This logic is captured in
% the rsiRebalanceFcn function defined in the Local Functions section.
rsiInitWeights = computeInitialWeights(rsiSignal(20,:));
rsiStrategy = backtestStrategy('RSI',@rsiRebalanceFcn,...
    'TransactionCosts',tradingCosts,...
    'LookbackWindow',2,...
    'InitialWeights',rsiInitWeights);
```

Set Up Backtest

As a benchmark, this example also runs a simple equal-weighted strategy to determine if the trading signals are providing valuable insights into future returns of the assets. The benchmark strategy is rebalanced every four weeks.

```
% The equal weight strategy requires no history, so set LookbackWindow to 0.
benchmarkStrategy = backtestStrategy('Benchmark',@equalWeightFcn,...
    'TransactionCosts',tradingCosts,...
    'RebalanceFrequency',20,...
    'LookbackWindow',0);
```

Aggregate each of the individual signal timetables into a single backtest signal timetable.

```
% Combine the three signal timetables.
signalTT = timetable;
signalTT = synchronize(signalTT, smaSignal);
signalTT = synchronize(signalTT, macdSignal);
signalTT = synchronize(signalTT, rsiSignal);
```

Use `backtestEngine` to create the backtesting engine and then use `runBacktest` to run the backtest. The risk-free rate earned on uninvested cash is 1% annualized.

```
% Put the benchmark strategy and three signal strategies into an array.
strategies = [benchmarkStrategy smaStrategy macdStrategy rsiStrategy];
% Create the backtesting engine.
bt = backtestEngine(strategies,'RiskFreeRate',0.01)
```

```
bt = 
  backtestEngine with properties:

               Strategies: [1x4 backtestStrategy]
             RiskFreeRate: 0.0100
           CashBorrowRate: 0
          RatesConvention: "Annualized"
                    Basis: 0
    InitialPortfolioValue: 10000
           DateAdjustment: "Previous"
                NumAssets: []
                  Returns: []
                Positions: []
                 Turnover: []
                  BuyCost: []
                 SellCost: []
                     Fees: []
```

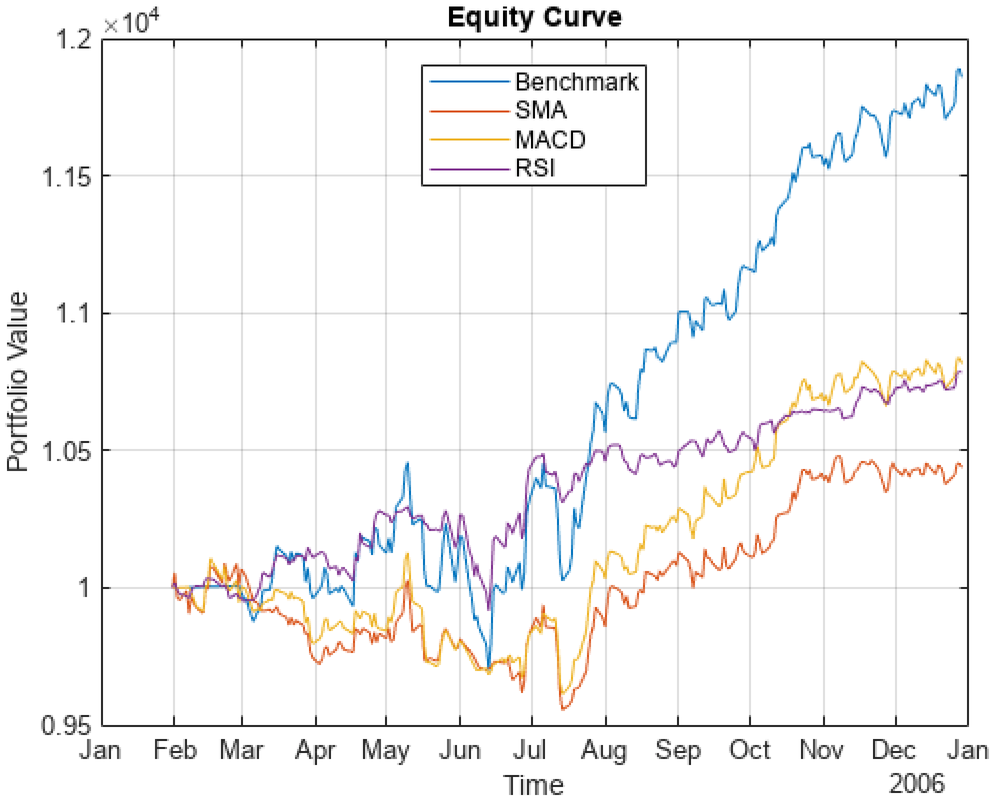
Backtest Strategies

```
% Start with the end of the initial weights calculation warm-up period.
startIdx = 20;

% Run the backtest.
bt = runBacktest(bt,pricesTT,signalTT,'Start',startIdx);
```

Examine Backtest Results

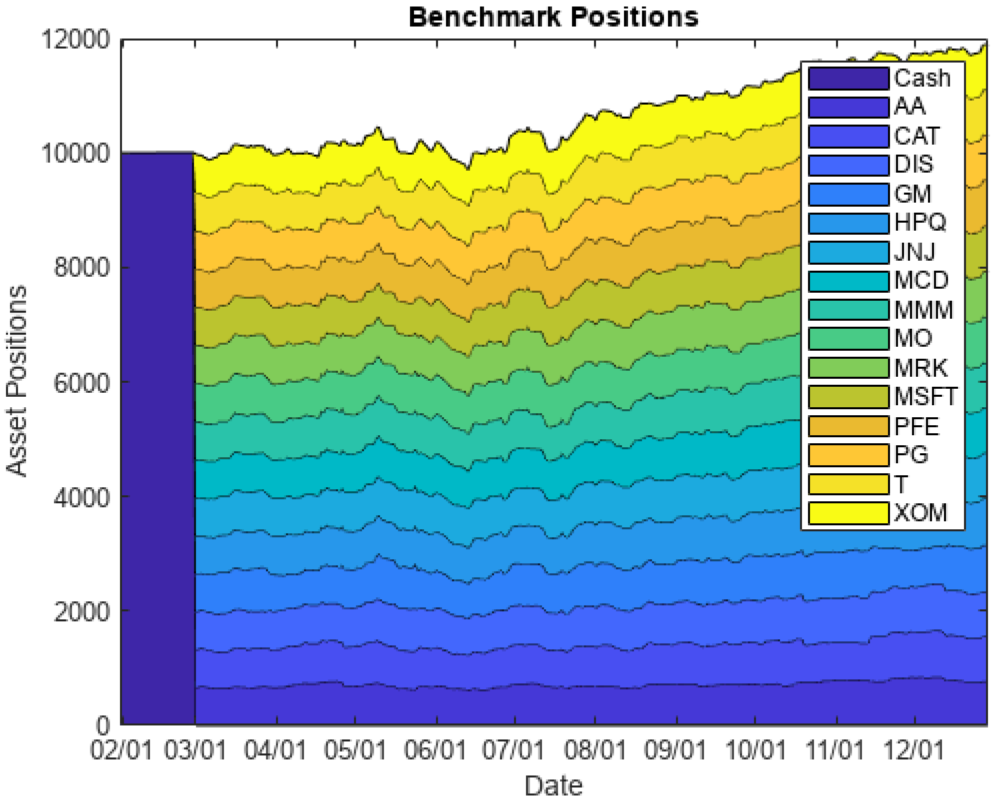
Use `equityCurve` to plot the strategy equity curves to visualize their performance over the backtest.

```
equityCurve(bt)
```



As mentioned previously, these strategies are not typically used as standalone trading signals. In fact, these three strategies perform worse than the simple benchmark strategy for the 2006 timeframe. You can visualize how the strategy allocations change over time using an area chart of the daily asset positions. To do so, use the `assetAreaPlot` helper function, defined in the Local Functions on page 4-254 section.

```
strategyName = Equal Weighted ;
assetAreaPlot(bt,strategyName)
```



Conclusion

The broad equity market had a very bullish 6 months in the second half of 2006 and all three of these strategies failed to fully capture that growth by leaving too much capital in cash. While none of these strategies performed well on their own, this example demonstrates how you can build signal-based trading strategies and backtest them to assess their performance.

Local Functions

The initial weight calculation function as well as the strategy rebalancing functions follow.

```
function initial_weights = computeInitialWeights(signals)
% Compute initial weights based on most recent signal.

nAssets = size(signals,2);
final_signal = signals{end,:};
buys = final_signal == 1;
initial_weights = zeros(1,nAssets);
initial_weights(buys) = 1 / nAssets;

end

function new_weights = crossoverRebalanceFcn(current_weights, pricesTT, signalTT, signalNameEndi
% Signal crossover rebalance function.

% Build cell array of signal names that correspond to the crossover signals.
symbols = pricesTT.Properties.VariableNames;
```

```
signalNames = cellfun(@(s) sprintf('%s%s',s,signalNameEnding), symbols, 'UniformOutput', false);

% Pull out the relevant signal data for the strategy.
crossoverSignals = signalTT(:,signalNames);

% Start with our current weights.
new_weights = current_weights;

% Sell any existing long position where the signal has turned to 0.
idx = crossoverSignals{end,:} == 0;
new_weights(idx) = 0;

% Find the new crossovers (signal changed from 0 to 1).
idx = crossoverSignals{end,:} == 1 & crossoverSignals{end-1,:} == 0;

% Bet sizing, split available capital across all remaining assets, and then
% invest only in the new positive crossover assets.  This leaves some
% proportional amount of capital uninvested for future investments into the
% zero-weight assets.
availableCapital = 1 - sum(new_weights);
uninvestedAssets = sum(new_weights == 0);
new_weights(idx) = availableCapital / uninvestedAssets;

end

function new_weights = rsiRebalanceFcn(current_weights, pricesTT, signalTT)
% Buy and sell on 1 and -1 rebalance function.

signalNameEnding = '_RSI';

% Build cell array of signal names that correspond to the crossover signals.
symbols = pricesTT.Properties.VariableNames;
signalNames = cellfun(@(s) sprintf('%s%s',s,signalNameEnding), symbols, 'UniformOutput', false);

% Pull out the relevant signal data for the strategy.
buySellSignals = signalTT(:,signalNames);

% Start with the current weights.
new_weights = current_weights;

% Sell any existing long position where the signal has turned to -1.
idx = buySellSignals{end,:} == -1;
new_weights(idx) = 0;

% Find the new buys (signal is 1 and weights are currently 0).
idx = new_weights == 0 & buySellSignals{end,:} == 1;

% Bet sizing, split available capital across all remaining assets, and then
% invest only in the new positive crossover assets.  This leaves some
% proportional amount of capital uninvested for future investments into the
% zero-weight assets.
availableCapital = 1 - sum(new_weights);
uninvestedAssets = sum(new_weights == 0);
new_weights(idx) = availableCapital / uninvestedAssets;

end
```

```
function new_weights = equalWeightFcn(current_weights,~)
% Equal-weighted portfolio allocation.

nAssets = numel(current_weights);
new_weights = ones(1,nAssets);
new_weights = new_weights / sum(new_weights);

end

function assetAreaPlot(backtester,strategyName)
% Plot the asset allocation as an area plot.

t = backtester.Positions.(strategyName).Time;
positions = backtester.Positions.(strategyName).Variables;
h = area(t,positions);
title(sprintf('%s Positions',strrep(strategyName,'_',' ')));
xlabel('Date');
ylabel('Asset Positions');
datetick('x','mm/dd','keepticks');
xlim([t(1) t(end)])
oldylim = ylim;
ylim([0 oldylim(2)]);
cm = parula(numel(h));
for i = 1:numel(h)
    set(h(i),'FaceColor',cm(i,:));
end
legend(backtester.Positions.(strategyName).Properties.VariableNames)

end
```

See Also

`backtestStrategy` | `backtestEngine` | `runBacktest` | `summary`

Related Examples

- “Backtest Investment Strategies Using Financial Toolbox™” on page 4-231
- “Backtest Strategies Using Deep Learning” on page 4-298
- “Backtest with Brinson Attribution to Evaluate Portfolio Performance” on page 4-311

Portfolio Optimization Using Social Performance Measure

Use a `Portfolio` object to minimize the variance, maximize return, and maximize the average percentage of women on a company's board. The same workflow can be applied with other Environmental, Social and Governance (ESG) criteria, such as an ESG score, a climate, or a temperature score.

The goal of this example is to find portfolios that are effcient in the sense that they minimize the variance, maximize return, and maximize the average percentage of women on the board of directors. To find the average percentage of women on a company's board (`WoB`) for a given portfolio, this example uses a weighted sum of the percentages of `WoB` for each individual asset, where the weights are given by the amount invested in each asset for the portfolio. By defining the average percentage of `WoB` this way, the `WoB` function is linear with respect to the weights.

Load Portfolio Data

```
load CAPMuniverse
% Assume that the percentage of women on the board of directors per
% company are as follows
WoB = [0.2857; 0.5; 0.3; 0.2857; 0.3077; 0.2727; ...
    0.4167; 0.2143; 0.3; 0.4167; 0.3077];
table(WoB,'VariableNames',{'WoB'},'RowNames',Assets(1:11))
```

```
ans=11×1 table
               WoB  
             ______

    AAPL     0.2857
    AMZN        0.5
    CSCO        0.3
    DELL     0.2857
    EBAY     0.3077
    GOOG     0.2727
    HPQ      0.4167
    IBM      0.2143
    INTC        0.3
    MSFT     0.4167
    ORCL     0.3077
```

Create `Portfolio` Object

Create a standard `Portfolio` object and incorporate the list of assets and estimate the moments of the assets' returns from the data. Use `setDefaultConstraints` to set the default mean-variance portfolio constraints. These constraints require fully invested, long-only portfolios where the nonnegative weights must sum to `1`.

```
% Create portfolio with default constraints
p = Portfolio('AssetList',Assets(1:11));
p = estimateAssetMoments(p, Data(:,1:11));
p = setDefaultConstraints(p);
```

Set Group Constraints

Use getGroups to include group contraints. The first group constraint ensures that the weights invested in mixed retail (Amazon and eBay) are at least 15%. The second group constraint ensures that the weights invested in computer companies (Apple, Dell and HP) are between 25% and 50%.

```
% Group constraints
G = [0 1 0 0 1 0 0 0 0 0 0;
     1 0 0 1 0 0 1 0 0 0 0];
LowG = [0.15; 0.25];
UpG = [Inf; 0.5];
p = setGroups(p, G, LowG, UpG);
```

Find the minimum and maximum percentage of WoB that a portfolio can attain given these extra group constraints. This is done using the estimateCustomObjectivePortfolio function with the average percentage of women on a company's board as the objective function.

```
% Set average WoB as the objective
objFun = @(x) WoB'*x;
```

Find the portfolio with the minimum average percentage of WoB with the group constraints using estimateCustomObjectivePortfolio with the function handle objFun.

```
% Minimum percetage of women on the board
wgt_minWoB = estimateCustomObjectivePortfolio(p,objFun);
minWoB = objFun(wgt_minWoB)
```

```
minWoB = 0.2462
```

Find the portfolio with the maximum average percentage of WoB with the group constraints using estimateCustomObjectivePortfolio with the name-value argument ObjectiveSense set to maximize.

```
% Maximum percentage of women on the board
wgt_maxWoB = estimateCustomObjectivePortfolio(p,objFun,...
    ObjectiveSense="maximize");
maxWoB = objFun(wgt_maxWoB)
```

```
maxWoB = 0.4792
```

Compute and Plot the Efficient Surface

Define a grid of WoB percentages such that minWoB = targetWoB(1) ≤ ... ≤ targetWoB(N) = maxWoB.

```
N = 20; % Size of grid
targetWoB = linspace(minWoB,maxWoB,N);
```

Use setInequality to set the percentage of WoB as a constraint. The coefficients of the linear constraint should be the WoB percentages associated to each asset, and the right-hand side should be the target portfolio WoB. The convention of the inequality is ≤. Since the goal is to maximize portfolio WoB, then the target WoB should be a lower bound for the portfolio WoB. Therefore, the signs of the coefficients and the right-hand side of the added inequality should be flipped.

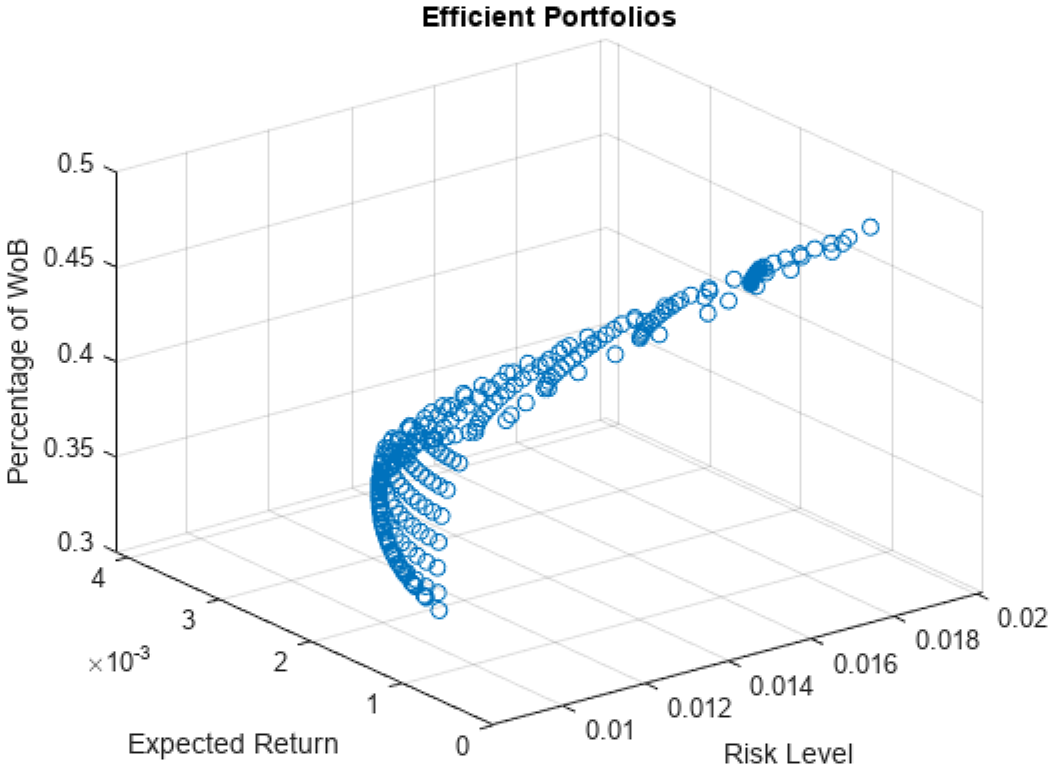
```
Ain = -WoB';
bin = -minWoB; % Start with the smallest WoB
p = setInequality(p,Ain,bin);
```

For each target WoB, targetWoB(i), find the efficient mean-variance frontier using estimateFrontier. At each iteration, the right-hand side of the WoB portfolio constraint should be changed to ensure that the returned portfolios achieve at least the target WoB. This method returns the weights of the portfolios on the mean-variance efficient frontier that have a WoB of at least targetWoB(i). Using the weights obtained for each target WoB, compute the portfolios' expected return, risk, and percentage of WoB.

```
% Get efficient surface values
prsk = cell(N,1);
pret = cell(N,1);
pWoB = cell(N,1);
for i = 1:N
    p.bInequality = -targetWoB(i);
    pwgt = estimateFrontier(p,N);
    [prsk{i},pret{i}] = estimatePortMoments(p,pwgt);
    pWoB{i} = pwgt'*WoB;
end
```

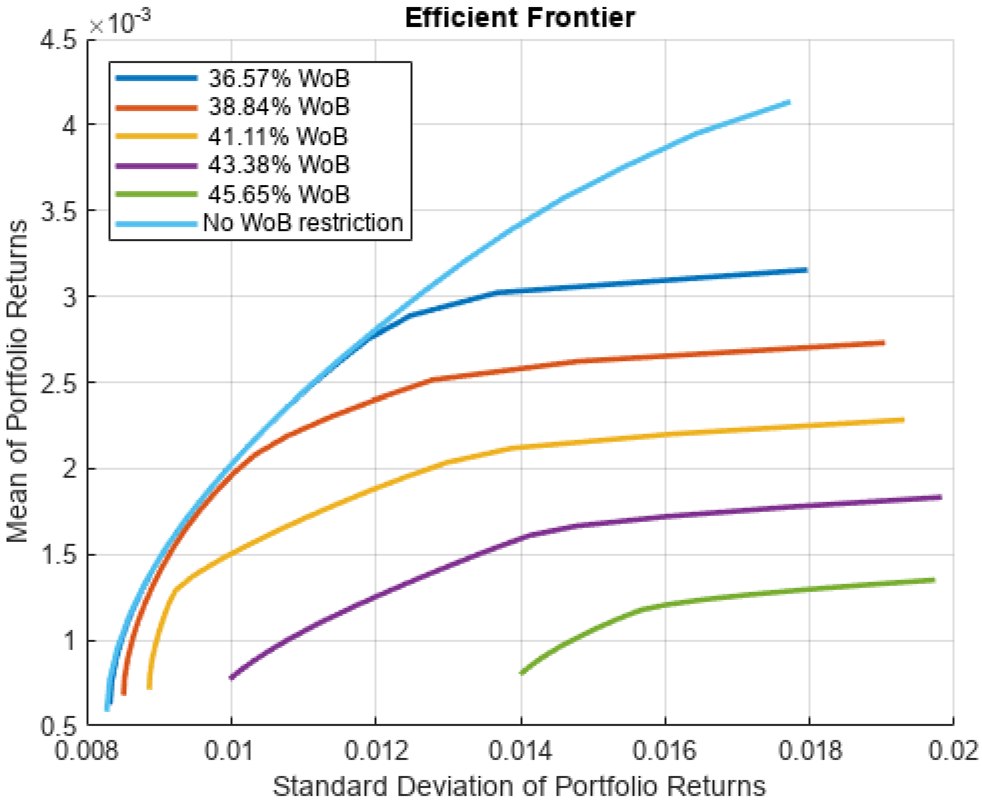
Plot the efficient portfolios.

```
% Plot efficient surface
scatter3(cell2mat(prsk),cell2mat(pret),cell2mat(pWoB))
title('Efficient Portfolios')
xlabel('Risk Level')
ylabel('Expected Return')
zlabel('Percentage of WoB')
```



To visualize the tradeoff between a portfolio's average percentage of `WoB` and the traditional mean-variance efficient frontier, a set of contour plots are computed for some target `WoB` percentages using the `plotContours` function in Local Functions on page 4-262.

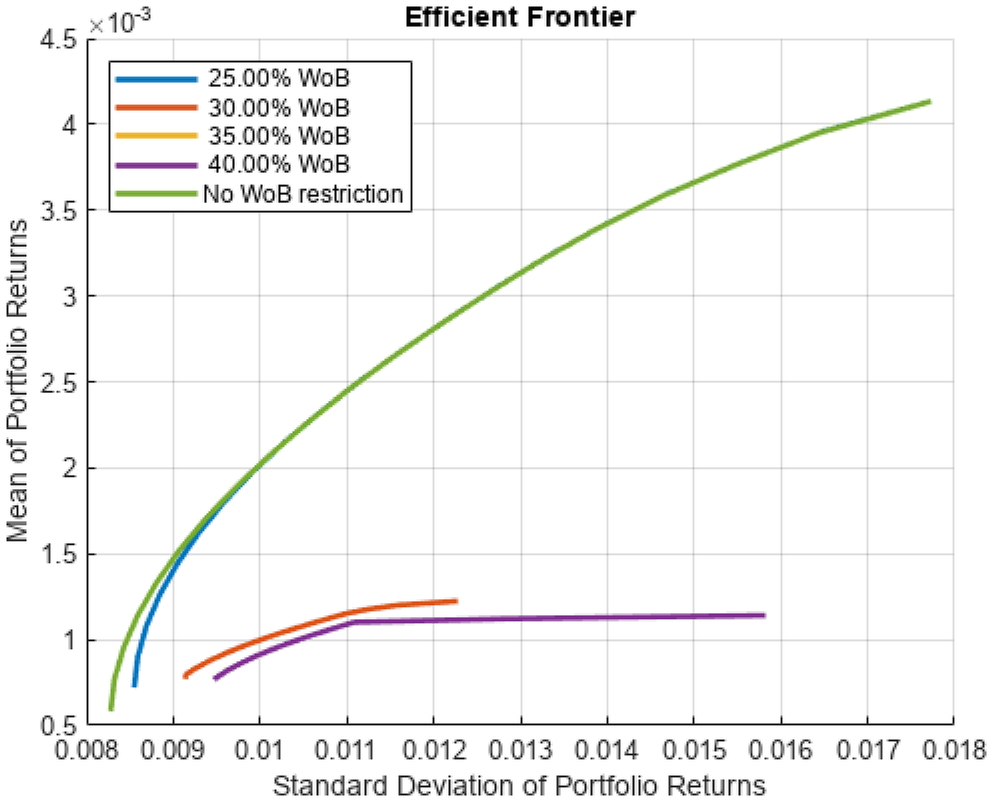
```
nC = 5; % Number of contour plots
minContour = max(pWoB{1}); % WoB values lower than this return
                           % overlapped contours.

% Plot contours
plotContours(p,minContour,maxWoB,nC,N)
```



Exclusion Examples

Instead of requiring a specific level for the portfolio's average percentage of `WoB`, the goal is to find the traditional mean-variance efficient frontiers while excluding assets that have a percentage of WoB lower than a given threshold. You can plot the exclusion using the `plotExclusionExample` function in Local Functions on page 4-262.

```
% Remove the average percetage of WoB constraint
p.AInequality = []; p.bInequality = [];

% Set of thresholds for excluding assets
thresholdWoB = 0.25:0.05:0.40;

% Plot exclusion example
plotExclusionExample(p,WoB,thresholdWoB,N)
```



The differences between this approach and the one presented in the previous section are quite evident. Requiring all the assets to have a `WoB` percentage of at least 35% gives an efficient frontier that can achieve a return of at most around $1.2(10)^{-3}$. On the other hand, requiring only that the portfolio's average percentage of `WoB` is 36.57% gives the possibility to reach a return of around $3.2(10)^{-3}$, almost 2.5 times the return obtained when excluding assets. To better show the diffences between these two approaches, compute the maximum return achieved for a given standard deviation for the two ways of including the percentage of `WoB` requirements to the portfolio.

Approach 1

In the first approach, exclude all assets with a `WoB` percentage lower than 33% and find the portfolio of maximum return that has a standard deviation of at most `0.012`.

```
% Select assets to exclude
ub = zeros(p.NumAssets,1);
ub(WoB >= 0.33) = 1;
p.UpperBound = ub;
% Estimate the return for a risk level of 0.012
pwgt_exclude = estimateFrontierByRisk(p,0.012);
ret_exclude = estimatePortReturn(p,pwgt_exclude)
```

```
ret_exclude = 0.0011
```

```
% Return constraints to the original portfolio
p.UpperBound = [];
```

Approach 2

For the second approach, ensure that the average `WoB` percentage is of at least 33% and find the portfolio of maximum return that has a standard deviation of at most `0.012`.

```
% Include WoB constraint into the portfolio
p = addInequality(p,-WoB',-0.33);
% Estimate the return for a risk level of 0.012
pwgt_avgWoB = estimateFrontierByRisk(p,0.012);
ret_avgWoB = estimatePortReturn(p,pwgt_avgWoB)
```

```
ret_avgWoB = 0.0028
```

```
% Return constraints to the original portfolio
p.AInequality = []; p.bInequality = [];
```

Compute the increase in return between these two approaches.

```
ret_increase = (ret_avgWoB-ret_exclude)/ret_exclude
```

```
ret_increase = 1.5202
```

This `ret_increase` value shows that the return from the approach that only bounds the portfolio's average WoB percentage instead of excluding certain assets has a return 152% higher (for the same risk level). Hence, when tackling problems with more than two objectives, excluding assets that do not meet a certain criteria might not be the best option. Instead, a weighted sum of the criteria of interest might show better results.

Local Functions

```
function [] = plotContours(p,minWoB,maxWoB,nContour,nPort)

% Set of WoB levels for contour plot
contourWoB = linspace(minWoB,maxWoB,nContour+1);

% Compute and plot efficient frontier for each value in contourWoB
figure;
hold on
labels = strings(nContour+1,1);
for i = 1:nContour
    p.bInequality = -contourWoB(i);
    plotFrontier(p,nPort);
    labels(i) = sprintf("%6.2f%% WoB",contourWoB(i)*100);
end
% Plot the "original" mean-variance frontier, i.e., the frontier
% without WoB requierements
p.AInequality = []; p.bInequality = [];
plotFrontier(p,nPort);
labels(i+1) = "No WoB restriction";
legend(labels,'Location','northwest')
hold off

end

function [] = plotExclusionExample(p,WoB,thresholdWoB, ...
    nPort)

% Compute and plot efficient frontier excluding assets that are below
% the WoB threshold
nT = length(thresholdWoB);
figure;
hold on
labels = strings(nT+1,1);
for i=1:nT
    ub = zeros(p.NumAssets,1);
    % Only select assets above WoB threshold
    ub(WoB >= thresholdWoB(i)) = 1;
    p.UpperBound = ub;
    plotFrontier(p,nPort);
    labels(i) = sprintf("%6.2f%% WoB",thresholdWoB(i)*100);
end
% Plot the "original" mean-variance frontier, i.e., the frontier
% without the WoB threshold
p.UpperBound = [];
plotFrontier(p,nPort);
labels(i+1) = "No WoB restriction";
legend(labels,'Location','northwest')
hold off

end
```

See Also

Portfolio | setBounds | addGroups | setAssetMoments | estimateAssetMoments | estimateBounds | plotFrontier | estimateFrontierLimits | estimateFrontierByRisk | estimatePortRisk

Related Examples

- “Creating the Portfolio Object” on page 4-24
- “Working with Portfolio Constraints Using Defaults” on page 4-57
- “Validate the Portfolio Problem for Portfolio Object” on page 4-90
- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-94
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-118
- “Postprocessing Results to Set Up Tradable Portfolios” on page 4-126
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-183
- “Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215
- “Portfolio Optimization Against a Benchmark” on page 4-195
- “Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
- “Diversify Portfolios Using Custom Objective” on page 4-329

More About

- “Portfolio Object” on page 4-19
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-17

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Diversify ESG Portfolios

This example shows how to include qualitative factors for environmental, social, and corporate governance (ESG) in the portfolio selection process. The example extends the traditional mean-variance portfolio using a Portfolio object to include the ESG metric. First, the estimateFrontier function computes the mean-variance efficient frontier for different ESG levels. Then, the example illustrates how to combine the ESG performance measure with portfolio diversification techniques. Specifically, it introduces hybrid models that use the Herfindahl-Hirshman (HH) index and the most diversified portfolio (MDP) approach using the estimateCustomObjectivePortfolio function. Finally, the backtestEngine framework compares the returns and behavior of the different ESG strategies.

Define Mean-Variance Portfolio

Load the table with the asset returns and a vector with ESG scores for the assets. Both the asset returns and their ESG scores are simulated values and do not represent the performance of any real securities. Hower, you can apply the code and workflow in this example to any data set with prices and returns and ESG information.

```
% Load data
load('asset_return_100_simulated.mat') % Returns table
load('ESG_s26.mat')                    % ESG scores
```

Transform the returns table into a prices timetable.

```
% Transform returns to prices
assetPrices = ret2tick(stockReturns);
% Transform prices table to timetable
nRows = size(stockReturns,1);
day = datetime("today");
Time = (day-nRows):day;
assetPrices = table2timetable(assetPrices,"RowTimes",Time);
```

Define a Portfolio object with default constraints where the weights must be nonnegative and sum to 1.

```
% Create a mean-variance Portfolio object with default constraints
p = Portfolio;
p = estimateAssetMoments(p,stockReturns);
p = setDefaultConstraints(p);
```

Compute the Mean-Variance Efficient Frontier for Different ESG Levels

Obtain contour plots of the ESG-mean-variance efficient surface. You obtain the efficient surface from the Pareto optima of the multiobjective problem that includes all the performance metrics: average ESG score, average return, and return variance.

First, obtain feasible values of the ESG metric by finding the minimum and maximum ESG levels. To do this step, use the estimateCustomObjectivePortfolio function assigning the average ESG score as the objective function. The average ESG score of a portfolio is the weighted sum of the individual asset ESG scores, where the weights are given by the amount invested in each asset.

```
% Define objective function: average ESG score
ESGscore = @(x) ESGnumeric'*x;
```

```
% Find the minimum ESG score
solMin = estimateCustomObjectivePortfolio(p,ESGscore);
minESG = ESGscore(solMin)
```

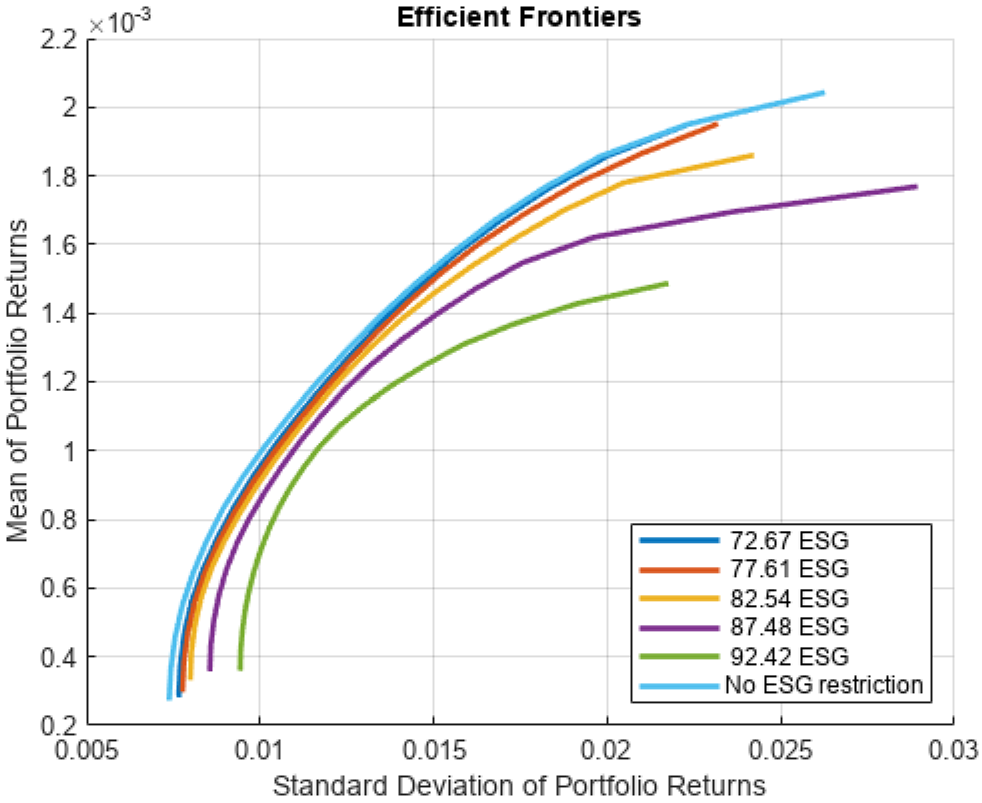
```
minESG = 0.0187
```

```
% Find the maximum ESG score
solMax = estimateCustomObjectivePortfolio(p,ESGscore, ...
    ObjectiveSense="maximize");
maxESG = ESGscore(solMax)
```

```
maxESG = 0.9735
```

To compute the contours of the efficient surface, you add the average ESG score as a constraint using the `setInequality` function. The coefficients of the linear constraint are the ESG scores associated with each asset and the right-hand side is the target ESG score for the desired contour. Notice that the convention of the inequalities in the `Portfolio` object is ≤. Because the goal is to maximize the average ESG score, the target ESG value for the contour should be a lower bound. Therefore, you need to flip the signs of the coefficients and the right-hand side of the added inequality. The function that computes and plots the contours is in the Local Functions on page 4-276 section.

```
N = 20; % Number of efficient portfolios per ESG value

% Add ESG score as a constraint to the Portfolio object
Ain = -ESGnumeric';
bin = -minESG; % Start with the smallest ESG score
p = setInequality(p,Ain,bin);

% Esimate lower value for contour plots
pwgt = estimateFrontier(p,N);
pESG = pwgt'*ESGnumeric;
minContour = max(pESG); % All ESG scores lower than this have
                        % overlapped contours.

% Plot contours
nC = 5; % Number of contour plots
plotESGContours(p,ESGnumeric,minContour,maxESG,nC,N);
```



Diversification Techniques

In this example, the two diversification measures are the equally weighted (EW) portfolio and the most diversified portfolio (MDP).

You obtain the EW portfolio using the Herfindahl-Hirschman (HH) index as the diversification measure

$$\mathrm{HH}(x) = \sum_{i=1}^{n} x_i^2$$

An equally weighted portfolio minimizes this index. Therefore, the portfolios that you obtain from using this index as a penalty have weights that satisfy the portfolio constraints and are more evenly weighted.

The diversification measure for the most diversified portfolio (MDP) is

$$\mathrm{MDP}(x) = -\sum_{i=1}^{n} \sigma_i x_i$$

where σ_i represents the standard deviation of asset i. Using this measure as a penalty function maximizes the diversification ratio [1 on page 4-276].

$$\varphi(x) = \frac{x^T \sigma}{\sqrt{x^T \Sigma x}}$$

If the portfolio is fully invested in one asset, or if all assets are perfectly correlated, the diversification ratio $\varphi(x)$ is equal to 1. For all other cases, $\varphi(x) > 1$. Therefore, if $\varphi(x) \approx 1$, there is no diversification, so the goal is to maximize $\varphi(x)$. Unlike the HH index, the goal of MDP is not to obtain a portfolio whose weights are evenly distributed among all assets, but to obtain a portfolio whose selected (nonzero) assets have the same correlation to the portfolio as a whole.

Diversification for Fixed ESG Level

You can extend the traditional minimum variance portfolio problem subject to an expected return to inlcude the ESG metric by setting an ESG constraint for the problem. The purpose of the ESG constraint is to force the portfolio to achieve an ESG score greater than a certain target. Then, add a penalty term to the objective function to guide the problem toward more or less diversified portfolios. How diversified a portfolio is depends on your choice of the penalty parameter.

Choose an ESG level of `0.85` and an expected return of `0.001`.

```
% Minimum ESG and return levels
ESG0 = 0.85;
ret0 = 1e-3;
```

Currently, the portfolio problem assumes that the weights must be nonnegative and sum to 1. Add the requirement that the return of the portfolio is at least `ret0` and the ESG score is at least `ESG0`. The feasible set is represented as X, which is

$$X = \left\{ x \;\middle|\; x \geq 0,\ \sum_{i=1}^{n} x_i = 1,\ \mu^T x \geq ret_0,\ \mathrm{ESG}(x) \geq ESG_0 \right\}$$

Add the ESG constraint using the `setInequality` function.

```
% Add ESG constraint
Ain = -ESGnumeric';
bin = -ESG0; % Set target ESG score
p = setInequality(p,Ain,bin);
```

To add the return constraint, pass the name-value argument `TargetReturn=ret0` to the `estimateCustomObjectivePortfolio` function for each of the different custom objective portfolios of interest.

The portfolio that minimizes the variance with the HH penalty is

$$\min_{x \in X} \; x^T \Sigma x + \lambda_{\mathrm{HH}} x^T x$$

```
% HH penalty parameter
lambdaHH = 0.001 ;
% Variance + Herfindahl-Hirschman (HH) index
var_HH = @(x) x'*p.AssetCovar*x + lambdaHH*(x'*x);
% Solution that accounts for risk and HH diversification
wHHMix = estimateCustomObjectivePortfolio(p,var_HH,TargetReturn=ret0);
```

The portfolio that minimizes the variance with the MDP penalty is

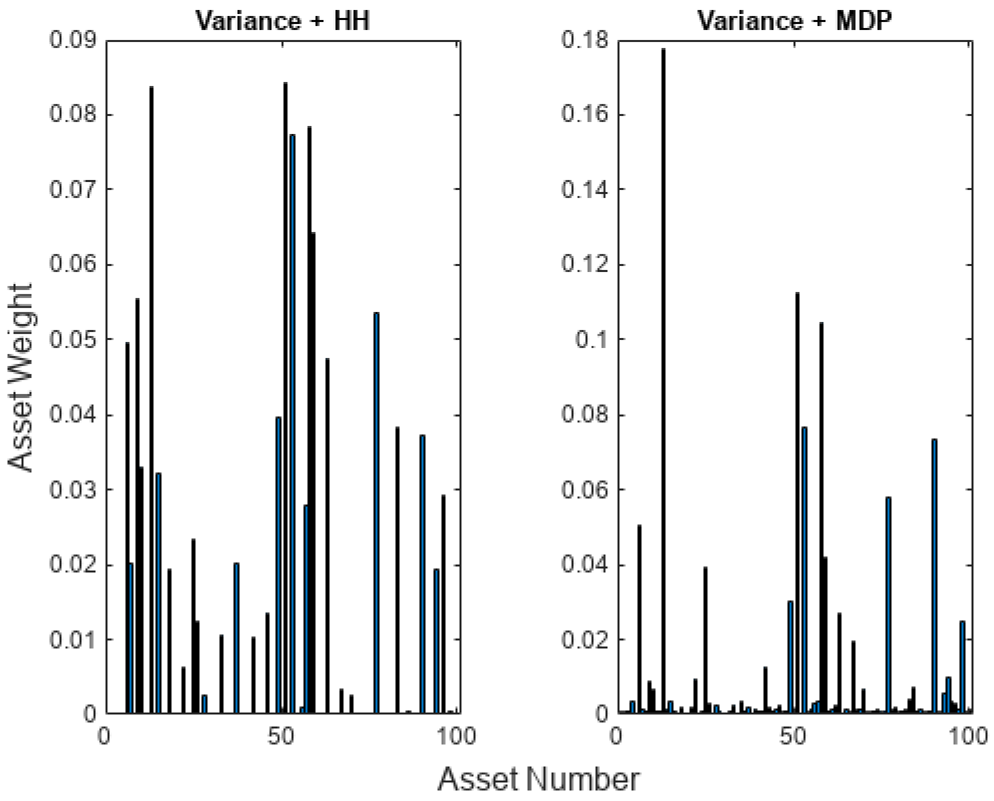
$$\min_{x \in X} \; x^T \Sigma x - \lambda_{\mathrm{MDP}}\, \sigma^T x$$

```
% MDP penalty parameter
lambdaMDP = 0.01 ;
```

```
% Variance + Most Diversified Portfolio (MDP)
sigma = sqrt(diag(p.AssetCovar));
var_MDP = @(x) x'*p.AssetCovar*x - lambdaMDP*(sigma'*x);
% Solution that accounts for risk and MDP diversification
wMDPMix = estimateCustomObjectivePortfolio(p,var_MDP,TargetReturn=ret0);
```

Plot the asset allocation from the penalized strategies.

```
% Plot asset allocation
figure
t = tiledlayout(1,2);

% HH penalized method
nexttile
bar(wHHMix');
title('Variance + HH')

% MDP penalized method
nexttile
bar(wMDPMix')
title('Variance + MDP')

% General labels
ylabel(t,'Asset Weight')
xlabel(t,'Asset Number')
```



The strategies that include the penalty function in the objective give weights that are between the minimum variance portfolio weights and the weights from the respective maximum diversification technique. In fact, for the problem with the HH penalty, choosing $\lambda_{\mathrm{HH}} = 0$ returns the minimum variance solution, and as $\lambda_{\mathrm{HH}} \to \infty$, the solution approaches the one that maximizes HH diversification. For the problem with the MDP penalty, choosing $\lambda_{\mathrm{MDP}} = 0$ also returns the minimum variance solution, and there exists a value $\widehat{\lambda}_{\mathrm{MDP}}$ such that the MDP problem and the penalized version are equivalent. Consequently, values of $\lambda_{\mathrm{MDP}} \in \left[0, \widehat{\lambda}_{\mathrm{MDP}}\right]$ return asset weights that range from the minimum variance behavior to the MDP behavior.

Diversification by "Tilting"

The strategies in Diversification for Fixed ESG Level on page 4-268 explicitly set a target ESG average score. However, a different set of strategies controls the ESG score in a less direct way. The method in these strategies uses ESG *tilting*. With tilting, you discretize the ESG score into `'high'` and `'low'` levels and the objective function penalizes each level differently. In other words, you use the diversification measure to tilt the portfolio toward higher or lower ESG values. Therefore, instead of explicitly requiring that the portfolios maintain a target ESG average score, you select assets, with respect to their ESG score, implicitly through the choice of penalty parameters.

Start by labeling the assets with an ESG score less than or equal to `0.5` as `'low'` and assets with an ESG score greater than `0.5` as `'high'`, and then remove the ESG constraint.

```
% Label ESG data
ESGlabel = discretize(ESGnumeric,[0 0.5 1], ...
    "categorical",{'low','high'});
% Create table with ESG scores and labels
ESG = table(ESGnumeric,ESGlabel);
% Remove ESG constraint
p = setInequality(p,[],[]);
```

The tilted version of the portfolio with the HH index is

$$\begin{aligned} &\min \;\; x^T \Sigma x + \lambda_{\mathrm{HH}}^{\mathrm{high}} \sum_{i \in H} x_i^2 + \lambda_{\mathrm{HH}}^{\mathrm{low}} \sum_{i \in L} x_i^2 \\ &\text{s.t.} \;\; \sum_{i=1}^{n} x_i = 1,\; \mu^T x \geq ret_0,\; x \geq 0 \end{aligned}$$

The feasible region does not bound the average ESG level of the portfolio. Instead, you implicitly control the ESG score by applying different penalization terms to assets with a `'high'` ESG score ($i \in H$) and assets with a `'low'` ESG score ($i \in L$). To achieve a portfolio with a high average ESG score, the penalty terms must satisfy that $0 \leq \lambda_{\mathrm{HH}}^{\mathrm{high}} \leq \lambda_{\mathrm{HH}}^{\mathrm{low}}$.

```
% HH penalty parameters
% Penalty parameter for assets with 'low' ESG score
lambdaLowHH =  0.01 ;
% Penalty parameter for assets with 'high' ESG score
lambdaHighHH =  0.001 ;
% Lambda for HH 'tilted' penalty
lambdaTiltHH = (ESG.ESGlabel=='low').*lambdaLowHH + (ESG.ESGlabel=='high').*lambdaHighHH;

% Variance + Herfindahl-Hirschman (HH) index
tilt_HH = @(x) x'*p.AssetCovar*x + lambdaTiltHH'*(x.^2);
% Solution that minimizes variance + HH term
wTiltHH = estimateCustomObjectivePortfolio(p,tilt_HH,TargetReturn=ret0);
```

Similarly, the tilted version of the portfolio with the MDP penalty term is given by

$$\begin{aligned}&\min\ \ x^T\Sigma x - \lambda_{\mathrm{MDP}}^{\mathrm{high}} \sum_{i\,\in\,H} \sigma_i x_i - \lambda_{\mathrm{MDP}}^{\mathrm{low}} \sum_{i\,\in\,L} \sigma_i x_i \\ &\mathrm{s.t.}\ \ \sum_{i\,=\,1}^{n} x_i = 1,\ \mu^T x \geq ret_0,\ x \geq 0\end{aligned}$$

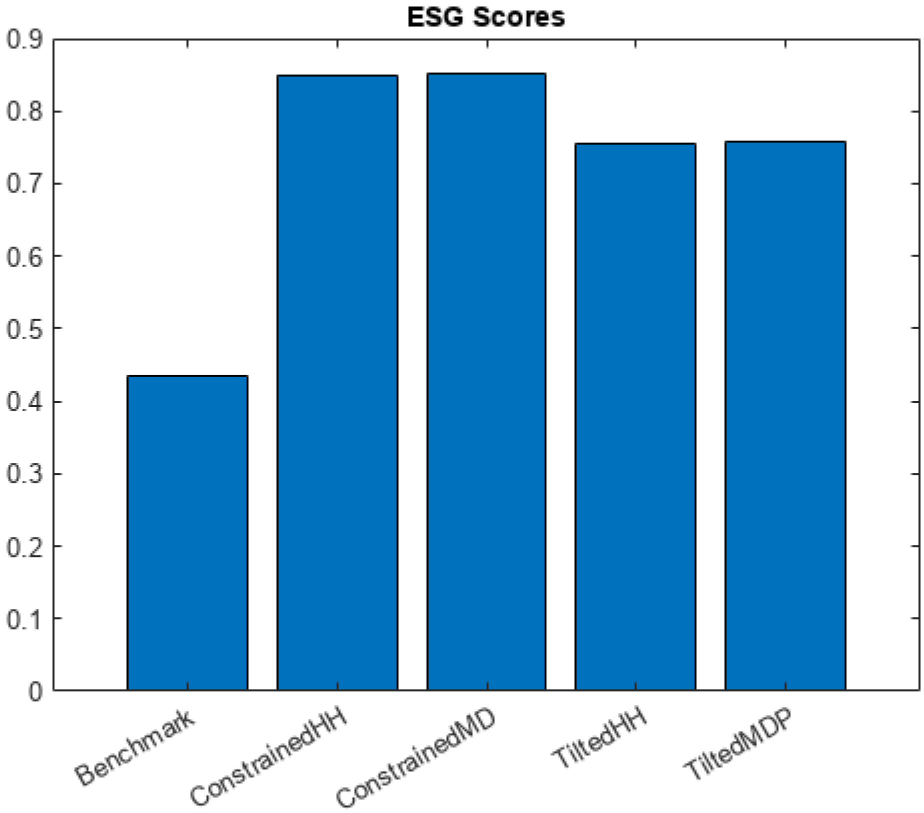
In this case, to achieve a portfolio with a high average ESG score, the penalty terms must satisfy that $0 \leq \lambda_{\mathrm{MDP}}^{\mathrm{low}} \leq \lambda_{\mathrm{MDP}}^{\mathrm{high}}$.

```
% MDP penalty parameters
% Penalty parameter for assets with 'low' ESG score
lambdaLowMDP = 0.001 ;
% Penalty parameter for assets with 'high' ESG score
lambdaHighMDP = 0.01 ;
% Lambda for MDP 'tilted' penalty
lambdaTiltMDP = (ESG.ESGlabel=='low').*lambdaLowMDP + (ESG.ESGlabel=='high').*lambdaHighMDP;

% Variance + MDP index
tilt_MDP = @(x) x'*p.AssetCovar*x - lambdaTiltMDP'*(sigma.*x);
% Solution that minimizes variance + HH term
wTiltMDP = estimateCustomObjectivePortfolio(p,tilt_MDP,TargetReturn=ret0);
```

Compare ESG Scores

Compare the ESG scores using tilting and a target ESG constraint. For the comparison to be more meaningful, compute the minimum variance portfolio without ESG constraints or penalty terms, and use the ESG score of the minimum variance portfolio as benchmark.

```
% Compute the minimum variance portfolio without ESG constraints
p = setInequality(p,[],[]);
wBmk = estimateFrontierByReturn(p,ret0);

% Compute ESG scores for the different strategies
Benchmark = ESGnumeric'*wBmk;
ConstrainedHH = ESGnumeric'*wHHMix;
ConstrainedMD = ESGnumeric'*wMDPMix;
TiltedHH = ESGnumeric'*wTiltHH;
TiltedMDP = ESGnumeric'*wTiltMDP;

% Create table
strategiesESG = table(Benchmark,ConstrainedHH,ConstrainedMD,TiltedHH,TiltedMDP);
figure;
bar(categorical(strategiesESG.Properties.VariableNames),strategiesESG.Variables)
title('ESG Scores')
```



As expected, the ESG scores of the penalized strategies are better than the ESG scores of the minimum variance portfolio without constraint and penalty terms. However, the tilted strategies achieve lower ESG scores than the ones that include the target ESG score as a constraint. This comparison shows the flexibility of the tilted strategy. If the ESG target score is not an essential requirement, then you can consider a tilting strategy.

Backtest Using Strategies

To show the performance through time for the two strategies (Diversification for Fixed ESG Level on page 4-268 and Diversification by "Tilting" on page 4-270), use the `backtestEngine` framework. Use `backtestStrategy` to compare the strategies with a prespecified target ESG score with the strategies that use ESG tilting.

```
% Store info to pass to ESG constrained strategies
Ain = -ESGnumeric';
bin = -ESG0; % Set target ESG score
conStruct.p = setInequality(p,-ESGnumeric',-ESG0);
conStruct.ret0 = ret0;
conStruct.lambdaHH = 0.01 ;
conStruct.lambdaMDP = 0.05 ;

% Store info to pass to ESG tilting strategies
tiltStruct.p = setInequality(p,[],[]);
tiltStruct.ret0 = ret0;
% HH tilting penalty parameters
% Penalty parameter for assets with 'low' ESG score
HHlambdaLow = 0.1;
% Penalty parameter for assets with 'high' ESG score
HHlambdaHigh = 0.01;
% Combined penalty terms for HH
tiltStruct.lambdaHH = (ESG.ESGlabel=='low').*HHlambdaLow + ...
    (ESG.ESGlabel=='high').*HHlambdaHigh;
% MDP tilting penalty parameters
% Penalty parameter for assets with 'low' ESG score
MDPlambdaLow = 0.005;
% Penalty parameter for assets with 'high' ESG score
MDPlambdaHigh = 0.05;
% Combined penalty terms for MDP
tiltStruct.lambdaMDP = (ESG.ESGlabel=='low').*MDPlambdaLow + ...
    (ESG.ESGlabel=='high').*MDPlambdaHigh;
```

Define the investment strategies that you want to use to make the asset allocation decisions at each investment period. For this example, four investment strategies are defined as input to `backtestStrategy`. The first two strategies require a minimum ESG score and the last two use the ESG tilting method.

```
% Define backtesting parameters
warmupPeriod = 84;       % Warmup period
rebalFreq = 42;          % Rebalance frequency
lookback  = [42 126];     % Lookback window
transactionCost = 0.001; % Transaction cost for trade
% Constrained variance + HH strategy
strat1 = backtestStrategy('MixedHH', @(w,P) MixHH(w,P,conStruct), ...
    'RebalanceFrequency', rebalFreq, ...
    'LookbackWindow', lookback, ...
    'TransactionCosts', transactionCost, ...
    'InitialWeights', wHHMix);
% Constrained variance + MDP
strat2 = backtestStrategy('MixedMDP', @(w,P) MixMDP(w,P,conStruct), ...
    'RebalanceFrequency', rebalFreq, ...
    'LookbackWindow', lookback, ...
    'TransactionCosts', transactionCost, ...
    'InitialWeights', wMDPMix);
% HH tilted strategy
strat3 = backtestStrategy('TiltedHH', @(w,P) tiltedHH(w,P,tiltStruct), ...
    'RebalanceFrequency', rebalFreq, ...
    'LookbackWindow', lookback, ...
    'TransactionCosts', transactionCost, ...
    'InitialWeights', wTiltHH);
% MDP tilted strategy
strat4 = backtestStrategy('TiltedMDP', @(w,P) tiltedMDP(w,P,tiltStruct), ...
    'RebalanceFrequency', rebalFreq, ...
    'LookbackWindow', lookback, ...
    'TransactionCosts', transactionCost, ...
    'InitialWeights', wTiltMDP);
% All strategies
strategies = [strat1,strat2,strat3,strat4];
```

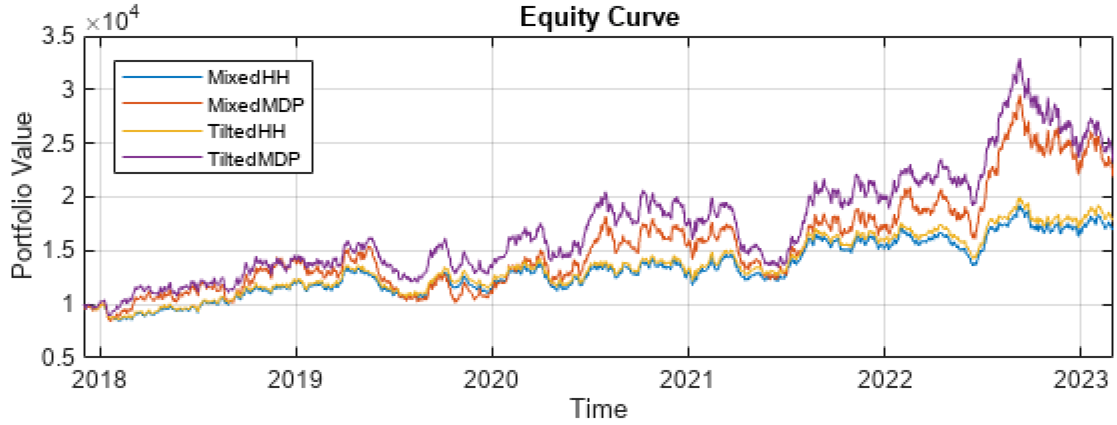
Run the backtest using `runBacktest` and generate a `summary` for each strategy's performance results.

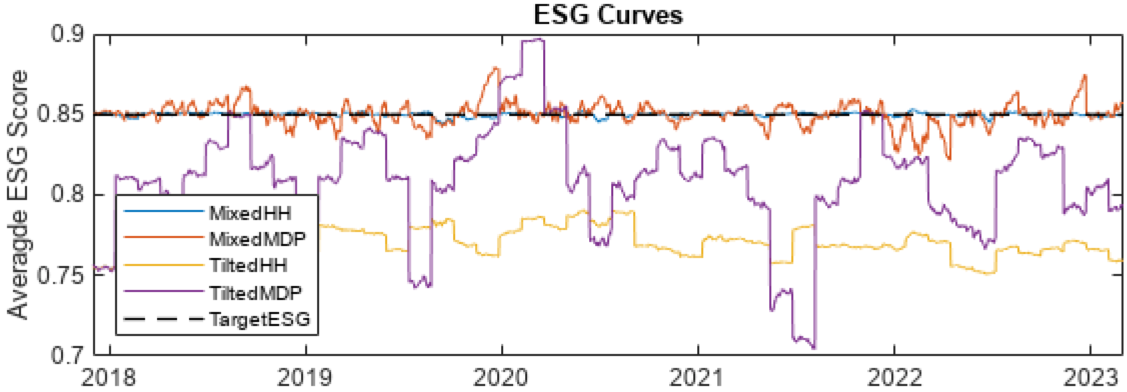
```
% Create the backtesting engine object
backtester = backtestEngine(strategies);
% Run backtest
backtester = runBacktest(backtester,assetPrices,...
    'Start',warmupPeriod);

% Summary
summary(backtester)
```

```
ans=9×4 table
                        MixedHH       MixedMDP      TiltedHH      TiltedMDP 
                       __________    __________    __________    __________

    TotalReturn            0.7077        1.1927       0.79527        1.3295
    SharpeRatio          0.029317       0.03188      0.031911      0.034608
    Volatility           0.011965      0.017845      0.011719      0.016851
    AverageTurnover     0.0054528     0.0065605     0.0053991     0.0070898
    MaxTurnover           0.80059       0.72624       0.80099       0.76676
    AverageReturn      0.00035068    0.00056875    0.00037387    0.00058304
    MaxDrawdown           0.23083       0.34334       0.22526       0.34688
    AverageBuyCost       0.068778      0.097277      0.070048       0.12019
    AverageSellCost      0.068778      0.097277      0.070048       0.12019
```

To visualize their performance over the entire investment period, plot the daily results of the strategies using `equityCurve`.

```
% Plot daily stocks and ESG strategy behavior
figure
t = tiledlayout(2,1,'Padding','none');

% Equity curves
nexttile(t)
equityCurve(backtester);

% Table and plot of the average ESG score for the mixed and tilted
% strategies throughout the investment period
nexttile(t)
TAvgESG = averageESGTimetable(backtester,ESGnumeric,ESG0);
```





In the Equity Curve plot, the tilted MDP strategy is the one that performs the best by the end of the investment period, followed by the mixed MDP strategy with the ESG constraint. The performance of the mixed and tilted strategies depend on the choice of the penalty parameters. Both ESG methods, the constrained and the tilted methods, require defining two parameters for each strategy. The ESG constrained method requires you to provide a target ESG score and a penalty parameter for the diversification term. The ESG tilting requires one penalty parameter value for the assets with `'high'` ESG scores and a different one for `'low'` ESG scores. In addition, the ESG tilting requires a third parameter to determine the cutoff point between `'high'` and `'low'` ESG assets. Given the dependency of the penalized strategies on the value of their parameters, the performance of those strategies vary widely. However, this example shows that it is possible to find values of the parameters so that the resulting strategies obtain good returns while improving on the average ESG score.

The ESG Curves plot shows the average ESG evolution computed throughout the investment period for the penalized investment strategies, both for the ESG constrained method and the ESG tilting method. You can see that, unlike the choice of the ESG target for the ESG constraint, the selection of the tilting penalty parameters has an effect on the ESG score of the optimal portfolios that is less explicit. Therefore, the average ESG score varies more with the tilting strategies than with the constrained strategies, as expected.

Although not shown in this example, the traditional minimum variance portfolio strategy, subject to the same expected return and ESG levels, results in higher average turnover and transaction costs than any of the strategies covered in the example. Another advanvange of adding a diversification measure to the formulation of the problem is to reduce the average turnover and transaction costs.

References

[1] Richard, J. C., and T. Roncalli. Smart Beta: Managing Diversification of Minimum Variance Portfolios. *Risk-Based and Factor Investing*. Elsevier, pp. 31-63, 2015.

Local Functions

```
function [] = plotESGContours(p,ESGscores,minESG,maxESG,nCont,...
    nPort)
% Plot mean-variance frontier for different ESG levels

% Add ESG constraint
p.AInequality = -ESGscores';

% Compute mean-variance risks and returns for different ESG levels
contourESG = linspace(minESG,maxESG,nCont+1);
figure
hold on
labels = strings(nCont+1,1);
for i = 1:nCont
    p.bInequality = -contourESG(i); % Change target ESG score
    plotFrontier(p,nPort);
    labels(i) = sprintf("%6.2f ESG",contourESG(i)*100);
end

% Plot the original mean-variance frontier
p.AInequality = []; p.bInequality = [];
plotFrontier(p,nPort);
labels(i+1) = "No ESG restriction";
title('Efficient Frontiers')
legend(labels,'Location','southeast')
hold off

end

function [ESGTimetable] = averageESGTimetable(backtester,...
    ESGscores,targetESG)
% Create a table of the average ESG score for the mix and tilted
% strategies throughout the investment period and plot it

% Normalize weights
wMixedHH = backtester.Positions.MixedHH{:,2:end}./...
    sum(backtester.Positions.MixedHH.Variables,2);
wMixedMDP = backtester.Positions.MixedMDP{:,2:end}./...
    sum(backtester.Positions.MixedMDP.Variables,2);
wTiltedHH = backtester.Positions.TiltedHH{:,2:end}./...
    sum(backtester.Positions.TiltedHH.Variables,2);
wTiltedMDP = backtester.Positions.TiltedMDP{:,2:end}./...
    sum(backtester.Positions.TiltedMDP.Variables,2);

% Compute ESG scores for the different strategies
consHH_ESG = wMixedHH*ESGscores;
consMDP_ESG = wMixedMDP*ESGscores;
tiltedHH_ESG = wTiltedHH*ESGscores;
tiltedMDP_ESG = wTiltedMDP*ESGscores;

% Create timetable
ESGTimetable = timetable(backtester.Positions.TiltedHH.Time,...
    consHH_ESG,consMDP_ESG,tiltedHH_ESG,tiltedMDP_ESG);

% Plot ESG curves
plot(ESGTimetable.Time, ESGTimetable.Variables);
hold on
plot(ESGTimetable.Time, targetESG*ones(size(ESGTimetable.Time)),...
    'k--','LineWidth',1); % Plots target ESG scores
title('ESG Curves');
ylabel('Averagde ESG Score');
legend('MixedHH','MixedMDP','TiltedHH','TiltedMDP', 'TargetESG',...
    'Location','southwest');

end

function new_weights = MixHH(~, assetPrices, struct)
% Min variance + max HH diversification strategy

% Retrieve portfolio information
p = struct.p;
ret0 = struct.ret0;

% Define returns and covariance matrix
assetReturns = tick2ret(assetPrices);
p = estimateAssetMoments(p,assetReturns{:,:});

% Objective function: Variance + Herfindahl-Hirschman
% diversification term
%   min x'*Sigma*x + lambda*x'*x
objFun = @(x) x'*p.AssetCovar*x + struct.lambdaHH*(x'*x);

% Solve problem
% Solution that minimizes the variance + HH index
new_weights = estimateCustomObjectivePortfolio(p,objFun,...
    TargetReturn=ret0);

end

function new_weights = MixMDP(~, assetPrices, struct)
% Min variance + MDP diversification strategy

% Retrieve portfolio information
p = struct.p;
ret0 = struct.ret0;

% Define returns and covariance matrix
assetReturns = tick2ret(assetPrices);
p = estimateAssetMoments(p,assetReturns{:,:});
sigma = sqrt(diag(p.AssetCovar));

% Objective function: Variance + MDP diversification term
%   min x'*Sigma*x - lambda*sigma'*x
objFun = @(x) x'*p.AssetCovar*x - struct.lambdaMDP*(sigma'*x);

% Solve problem
% Solution that minimizes variance + MDP term
new_weights = estimateCustomObjectivePortfolio(p,objFun,...
    TargetReturn=ret0);
```

```
end

function new_weights = tiltedHH(~,assetPrices,struct)
% Tilted HH approach

% Retrieve portfolio information
p = struct.p;
ret0 = struct.ret0;
lambda = struct.lambdaHH;

% Define returns and covariance matrix
assetReturns = tick2ret(assetPrices);
p = estimateAssetMoments(p,assetReturns{:,:});

% Objective function: Variance + Herfindahl-Hirschman
% diversification term
%   min x'*Sigma*x + lambda*x'*x
objFun = @(x) x'*p.AssetCovar*x + lambda'*(x.^2);

% Solve problem
% Solution that minimizes variance + HH term
new_weights = estimateCustomObjectivePortfolio(p,objFun,...
    TargetReturn=ret0);

end

function new_weights = tiltedMDP(~,assetPrices,struct)
% Tilted MDP approach

% Retrieve portfolio information
p = struct.p;
ret0 = struct.ret0;
lambda = struct.lambdaMDP;

% Define returns and covariance matrix
assetReturns = tick2ret(assetPrices);
p = estimateAssetMoments(p,assetReturns{:,:});
sigma = sqrt(diag(p.AssetCovar));

% Objective function: Variance + MDP
%   min x'*Sigma*x - lambda*sigma'*x
objFun = @(x) x'*p.AssetCovar*x - lambda'*(sigma.*x);

% Solve problem
% Solution that minimizes variance + MDP term
new_weights = estimateCustomObjectivePortfolio(p,objFun,...
    TargetReturn=ret0);

end

% The API of the rebalance functions (MixHH, MixMDP, tiltedHH, and
% tiltedMDP) require a first input with the current weights. They
% are redundant for these strategies and can be ignored.
```

See Also

Portfolio | setBounds | addGroups | setAssetMoments | estimateAssetMoments | estimateBounds | plotFrontier | estimateFrontierLimits | estimateFrontierByRisk | estimatePortRisk

Related Examples

- "Creating the Portfolio Object" on page 4-24
- "Working with Portfolio Constraints Using Defaults" on page 4-57
- "Validate the Portfolio Problem for Portfolio Object" on page 4-90
- "Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object" on page 4-94
- "Estimate Efficient Frontiers for Portfolio Object" on page 4-118
- "Postprocessing Results to Set Up Tradable Portfolios" on page 4-126
- "Portfolio Optimization with Semicontinuous and Cardinality Constraints" on page 4-183
- "Black-Litterman Portfolio Optimization Using Financial Toolbox™" on page 4-215
- "Portfolio Optimization Against a Benchmark" on page 4-195
- "Portfolio Optimization Examples Using Financial Toolbox™" on page 4-152
- "Portfolio Optimization Using Social Performance Measure" on page 4-257
- "Diversify Portfolios Using Custom Objective" on page 4-329

More About

- "Portfolio Object" on page 4-19
- "Portfolio Optimization Theory" on page 4-3
- "Portfolio Object Workflow" on page 4-17

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Risk Budgeting Portfolio

This example shows how to use `riskBudgetingPortfolio` to create a risk budgeting portfolio and `portfolioRiskContribution` to compute the risk contribution of the assets in the portfolio.

Risk budgeting is a portfolio allocation strategy that focuses on the allocation of risk to define the weights of a portfolio. The risk budgeting portfolio is constructed by ensuring that the risk contribution of each asset to the portfolio overall risk matches a target risk budget.

A common risk budgeting porfolio is the risk parity or equal risk contribution portfolio. For risk parity portfolios, the goal is to ensure that all assets contribute equally to the portfolio risk, that is, the target risk budget is the same for all assets. Key advantages of the risk budgeting portfolio are stability and diversification of weights:

- **Stability of the allocations**: The weights of the risk parity portfolio are more robust to changes in the returns covariance than the weights of the mean-variance portfolio.
- **Diversification of the portfolio weights**: The risk parity portfolio keeps the portfolio from concentrating in just a few number of assets.

This example follows the example in Section 3.1.1 of Bruder and Roncalli.[1 on page 4-284]

Define the Asset Returns Covariance

Define the covariance matrix of the asset returns (`Sigma`). `Sigma` must be a positive semidefinite matrix.

```
% Define the returns covariance matrix.
sigma = [0.2; 0.21; 0.1];      % Assets risk
rho = 0.8;                     % Assets correlation
C = @(rho) [  1  rho  rho;
            rho    1  rho;
            rho  rho    1 ];    % Uniform correlation
Sigma = corr2cov(sigma,C(rho)); % Covariance
% Check that Sigma is positive semidefinite. All
% eigenvalues should be nonnegative (>= 0).
eig(Sigma)
```

```
ans = 3×1

    0.0026
    0.0084
    0.0831
```

Create Risk Budgeting Portfolio

Use `riskBudgetingPortfolio` with the covariance matrix (`Sigma`) and a vector of risk budgets (`budget`) to compute the long-only fully invested risk budgeting portfolio. When you omot the target risk budget, the function computes the risk parity portfolio.

```
% Define a risk budget.
budget = [0.5; 0.25; 0.25];
wRB = riskBudgetingPortfolio(Sigma,budget)
```

```
wRB = 3×1
```

```
    0.3912
    0.1964
    0.4124
```

Compute Risk Contribution for Each Asset

Use `portfolioRiskContribution` to check that the risk contribution of the portfolio obtained in `wRB` matches the target risk budget. By default, `portfolioRiskContribution` computes the relative risk contribution of a given portfolio (`wRB`) with respect to the provided covariance matrix (`Sigma`).

```
prc = portfolioRiskContribution(wRB,Sigma)
```

```
prc = 3×1

    0.5000
    0.2500
    0.2500
```

As expected, the risk contribution of each asset matches the target risk budget (`budget`). You can also use `portfolioRiskContribution` to compute the absolute risk contribution of each asset.

```
mrc = portfolioRiskContribution(wRB,Sigma,RiskContributionType="absolute")
```

```
mrc = 3×1

    0.0751
    0.0376
    0.0376
```

Compare Risk Budgeting Portfolios with Mean-Variance

While the allocation strategy of the risk budgeting portfolio focuses only on the risk, the allocation strategy of the mean-variance portfolio also makes use of the assets' return information.

```
% Define the returns mean.
mu = [0.08; 0.08; 0.05];
```

Initialize a `Portfolio` object with default constraints using `Sigma` and `mu`.

```
% Define the mean-variance portfolio.
p = Portfolio(AssetMean=mu,AssetCovar=Sigma);
p = setDefaultConstraints(p);
```

Compute the portfolio on the efficient frontier with a target volatility equal to the risk attained by the risk budgeting portfolio previously computed as `wRB`.

```
% Define the target volatility.
targetRisk = sqrt(wRB'*Sigma*wRB);
wMV = estimateFrontierByRisk(p,targetRisk)
```

```
wMV = 3×1

    0.3846
    0.2030
    0.4124
```

Assume that the correlation between the assets is `0.7` instead of `0.8`. Use the new covariance matrix (`Sigma7`) to compute the same mean-variance and risk budgeting portfolios.

```
% Update the covariance matrix.
Sigma7 = corr2cov(sigma,C(0.7));
% Compute the mean-variance portfolio.
p = Portfolio(AssetMean=mu,AssetCovar=Sigma7);
p = setDefaultConstraints(p);
wMV_rho7 = estimateFrontierByRisk(p,targetRisk)
```

```
wMV_rho7 = 3×1

    0.3841
    0.2600
    0.3559
```

```
% Compute the risk budgeting portfolio.
wRB_rho7 = riskBudgetingPortfolio(Sigma7,budget)
```

```
wRB_rho7 = 3×1

    0.3844
    0.1986
    0.4170
```

What if the correlation is `0.9` instead of `0.8` (`Sigma9`) and the risk of the second asset is `0.18` instead of `0.21`?

```
% Update the covariance matrix.
sigma2 = sigma;
sigma2(2) = 0.18;
Sigma9 = corr2cov(sigma2,C(0.9));
% Compute the mean-variance portfolio.
p = Portfolio(AssetMean=mu,AssetCovar=Sigma9);
p = setDefaultConstraints(p);
wMV_rho9 = estimateFrontierByRisk(p,targetRisk)
```

```
wMV_rho9 = 3×1

         0
    0.6612
    0.3388
```

```
% Compute the risk budgeting portfolio.
wRB_rho9 = riskBudgetingPortfolio(Sigma9,budget)
```

```
wRB_rho9 = 3×1

    0.3852
    0.2196
    0.3952
```

What if the original covariance (`Sigma`) is correct, but the mean of the first asset is `0.09` instead of `0.08`?

```
% Update the returns vector.
mu1 = mu;
mu1(1) = 0.09;
% Compute the mean-variance portfolio.
p = Portfolio(AssetMean=mu1,AssetCovar=Sigma);
p = setDefaultConstraints(p);
wMV_mu1 = estimateFrontierByRisk(p,targetRisk)
```

```
wMV_mu1 = 3×1

    0.5664
         0
    0.4336
```

```
% Compute the risk budgeting portfolio.
wRB_mu1 = riskBudgetingPortfolio(Sigma,budget)
```

```
wRB_mu1 = 3×1

    0.3912
    0.1964
    0.4124
```

The weights of the risk budgeting portolio are more stable than those of the mean-variance portfolio. When the change is in the expected returns, the risk budgeting portfolio remains the same as in the original problem. The risk budgeting portfolio accounts only for changes to the covariance matrix and a change in the vector of expected returns does not affect the weights allocation.

```
% Display a table of weights for the risk budgeting portfolios.
RBtable = table(wRB,wRB_rho7,wRB_rho9,wRB_mu1)
```

```
RBtable=3×4 table
      wRB      wRB_rho7    wRB_rho9    wRB_mu1
    _______    ________    ________    _______

    0.39123    0.38444     0.38521     0.39123
    0.19638    0.19857     0.21957     0.19638
    0.41239    0.41699     0.39522     0.41239
```

```
% Display a table of weights for the mean-variance portfolios.
MVtable = table(wMV,wMV_rho7,wMV_rho9,wMV_mu1)
```

```
MVtable=3×4 table
      wMV      wMV_rho7    wMV_rho9    wMV_mu1
    _______    ________    ________    _______

    0.38462    0.38411           0     0.56638
    0.20301    0.26003      0.6612           0
    0.41237    0.35586      0.3388     0.43362
```

The results on these tables show that the risk budgeting portfolio is more robust to changes in the parameters. The weights in the third and fourth column of the mean-variance portfolio table change so drastically that the portfolios stop investing in some assets altogether when the parameters change. The mean-variance portfolio tends to concentrate weights in a few number of assets, which decreases portfolio diversification.

References

[1] Bruder, B. and T. Roncalli. *Managing Risk Exposures Using the Risk Budgeting Approach*. January 2012. Available at SSRN: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=2009778.

See Also

`portfolioRiskContribution` | `riskBudgetingPortfolio` | `Portfolio` | `setDefaultConstraints` | `plotFrontier` | `estimateFrontierLimits` | `estimateFrontierByRisk` | `estimatePortRisk`

Related Examples

- “Backtest Using Risk-Based Equity Indexation” on page 4-285
- “Creating the Portfolio Object” on page 4-24
- “Working with Portfolio Constraints Using Defaults” on page 4-57
- “Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-94
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-118
- “Portfolio Optimization Against a Benchmark” on page 4-195
- “Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
- “Portfolio Optimization Using Social Performance Measure” on page 4-257
- “Diversify Portfolios Using Custom Objective” on page 4-329

More About

- “Portfolio Object” on page 4-19
- “Portfolio Optimization Theory” on page 4-3
- “Portfolio Object Workflow” on page 4-17

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- MATLAB for Advanced Portfolio Construction and Stock Selection Models (30 min 28 sec)

Backtest Using Risk-Based Equity Indexation

This example shows how to use backtesting with a risk parity or equal risk contribution strategy rebalanced approximately every month as a risk-based indexation. In this example, you use the backtesting engine (`backtestEngine`) to create the risk parity strategy, that all assets in the portfolio contribute equally to the risk of the portfolio at each rebalancing period.

To highlight advantages, the example compares the proposed risk-based indexation against a capitalization-weighted indexation. In contrast to a risk parity strategy, the assets of capitalization-weighted portfolios are weighted with respect to the market portfolio. One disadvatage of capitalization-weighted indexation is that it is a trend-following strategy that leads to bubble-risk exposure as the best performers represent a larger proportion of the market. Also, capitalization-weighted indexation can sometimes lead to weights concentratng in a sector. A risk-based indexation is an alternative investment strategy to avoid these issues of capitalization-weighted indexation.

Define Assets

```
% Read the table of daily adjusted close prices for 2006 DJIA stocks.
T = readtable('dowPortfolio.xlsx');
% Convert the table to a timetable.
pricesTT = table2timetable(T);
% Define the number of assets (not counting the market index).
nAssets = size(pricesTT,2)-1;
```

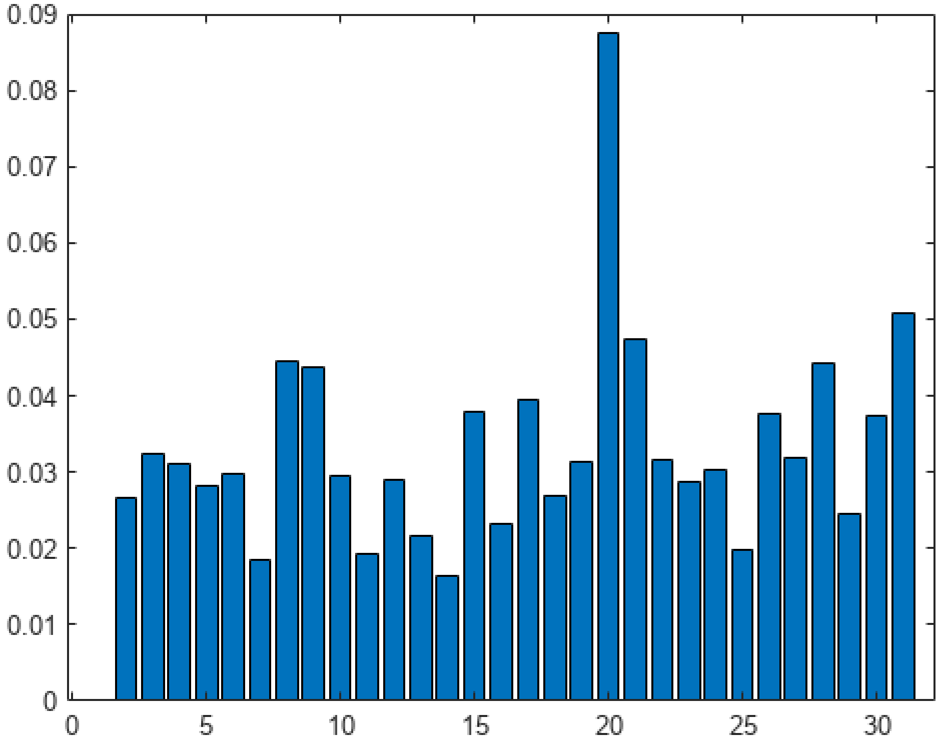
Define Risk Parity Strategy

To obtain the long-only fully invested risk parity portfolio, use the `riskBudgetingPortfolio` function. When you pass a returns covariance matrix to `riskBudgetingPortfolio`, the function computes the associated risk parity portfolio. To compute the initial weights for the risk parity strategy, `riskBudgetingPortfolio` uses the covariance matrix estimated with information from the first two months.

```
% Set backtesting warm-up period to two 21-day months.
warmupPeriod = 21*2;
% Set warm-up partition of timetable.
% Invest only in assets. First column of pricesTT is the market index.
warmupTT = pricesTT(1:warmupPeriod,2:end);
% Estimate the covariance matrix.
assetReturns = tick2ret(warmupTT);
assetCov = cov(assetReturns{:,:});
% Compute the initial risk parity portfolio.
initialRiskParity = [0; riskBudgetingPortfolio(assetCov)];
```

Visualize initial risk parity portfolio weights.

```
% Plot a bar chart.
bar(initialRiskParity)
```



Create the risk parity backtesting strategy using a `backtestStrategy` object. Set the risk parity strategy to rebalance every month.

```
% Rebalance frequency
rebalFreq = 21; % Every month
```

To gather enough data, set the lookback window to at least 2 months. To remove old data from the covariance estimation, set the lookback window to no more than 6 months.

```
% Lookback window
lookback = [42 126];
```

Use a fixed transaction cost equal to 0.5% of the amount traded.

```
% Fixed transaction cost
transactionCost = 0.005;
```

Define the risk parity allocation strategy using the `riskParityFcn` rebalancing function that is defined in Local Functions on page 4-289. Notice that the `riskBudgetingPortfolio` function is at the core of `riskParityFcn`.

```
% Risk parity allocation strategy
stratRP = backtestStrategy('Risk Parity', @riskParityFcn, ...
    RebalanceFrequency=rebalFreq, ...
    LookbackWindow=lookback, ...
    TransactionCosts=transactionCost, ...
    InitialWeights=initialRiskParity);
```

Define Benchmark Strategy

Next, compare the risk parity strategy against the portfolio that follows the Dow Jones Industrial Average (DJI) market index. To obtain the market-following portfolio, invest all assets to the first entry of the weights vector because the first column of the prices timetable (`pricesTT`) represents the market index.

```
% Compute the initial market portfolio.
initialMkt = [1;zeros(nAssets,1)];
```

Define the benchmark strategy using the `mktFcn` rebalancing function that is defined in Local Functions on page 4-289.

```
% Market index allocation strategy
stratBmk = backtestStrategy('Market Index', @mktFcn, ...
    RebalanceFrequency=rebalFreq, ...
    LookbackWindow=lookback, ...
    TransactionCosts=transactionCost, ...
    InitialWeights=initialMkt);
```

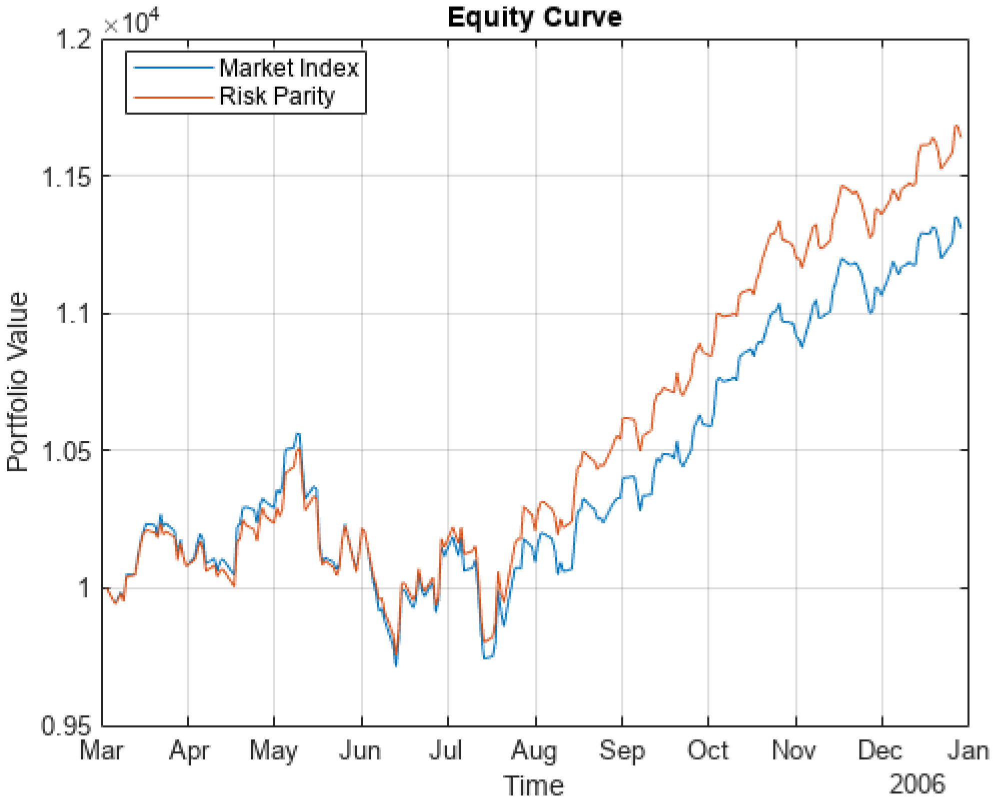
Run Backtest

Run the backtest using a `backtestEngine` object and `runBacktest`.

```
% Define strategies for backtest engine.
strategies = [stratBmk stratRP];
% Define a backtest engine.
backtester = backtestEngine(strategies);
% Run the backtest.
backtester = runBacktest(backtester,pricesTT,Start=warmupPeriod);
```

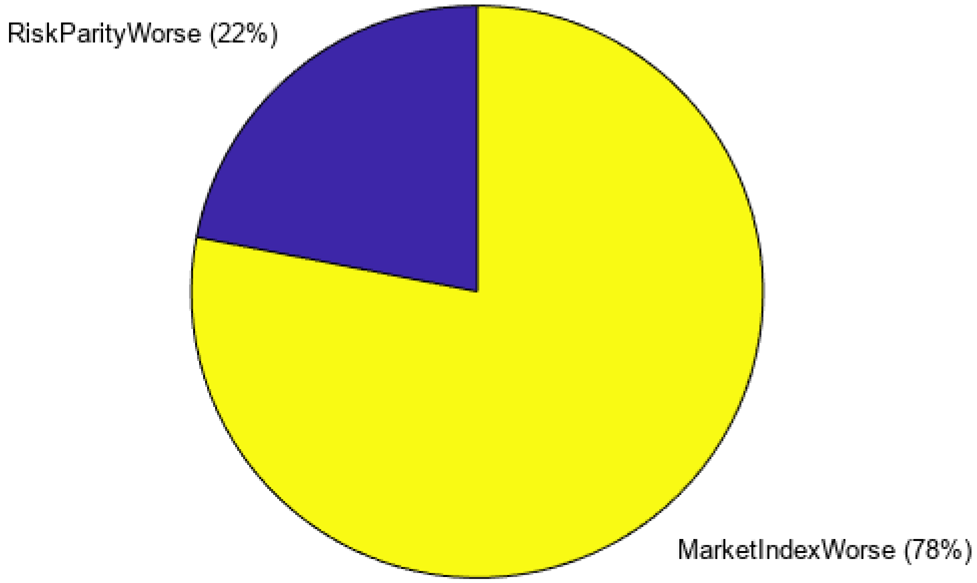
Use `equityCurve` to plot the equity curve.

```
% Plot equity curve.
equityCurve(backtester)
```



The risk parity strategy outperforms the market index by the end of the investment period. To understand why, you can look at which strategy performs better in the event of meaningful losses.

```
% Get returns timetable.
returnsTT = backtester.Returns;
% Determine levels of losses to consider.
cutoff = 0.004 ;

% Obtain scenarios with losses below cutoff.
idxLoss = returnsTT.Market_Index <= -cutoff | ...
    returnsTT.Risk_Parity <= -cutoff;
% Indices when risk parity outperforms the market
idxLossMarketWorse = ...
    returnsTT.Market_Index(idxLoss) <= returnsTT.Risk_Parity(idxLoss);
catLoss = categorical(idxLossMarketWorse, ...
    [0 1], {'RiskParityWorse','MarketIndexWorse'});
% Plot pie chart.
pie(catLoss)
```



The plot shows that in 78% of the scenarios with *meaningful losses*, the market index loses more than the risk parity strategy. In this example, the reason the risk parity strategy outperforms the market index is because the risk parity strategy is more resilient to larger losses.

Local Functions

```
function new_weights = riskParityFcn(~,pricesTT)
% Risk parity rebalance function

% Invest only in assets. First column of pricesTT is the market index.
assetReturns = tick2ret(pricesTT(:,2:end));
assetCov = cov(assetReturns{:,:});

% Do not invest in the market index.
new_weights = [0; riskBudgetingPortfolio(assetCov)];

end

function new_weights = mktFcn(~,pricesTT)
% Market index rebalance function

% Invest only in the market index.
new_weights = zeros(size(pricesTT,2),1);
new_weights(1) = 1;

end
```

```
% The API of the rebalance functions (riskParityFcn and mktFcn) require
% a first input with the current weights. They are
% redundant for these two strategies and can be ignored.
```

See Also

`backtestStrategy` | `backtestEngine` | `portfolioRiskContribution` | `riskBudgetingPortfolio`

Related Examples

- “Risk Budgeting Portfolio” on page 4-280
- “Create Hierarchical Risk Parity Portfolio” on page 4-291
- “Backtest Investment Strategies Using Financial Toolbox™” on page 4-231
- “Backtest Investment Strategies with Trading Signals” on page 4-244
- “Backtest with Brinson Attribution to Evaluate Portfolio Performance” on page 4-311

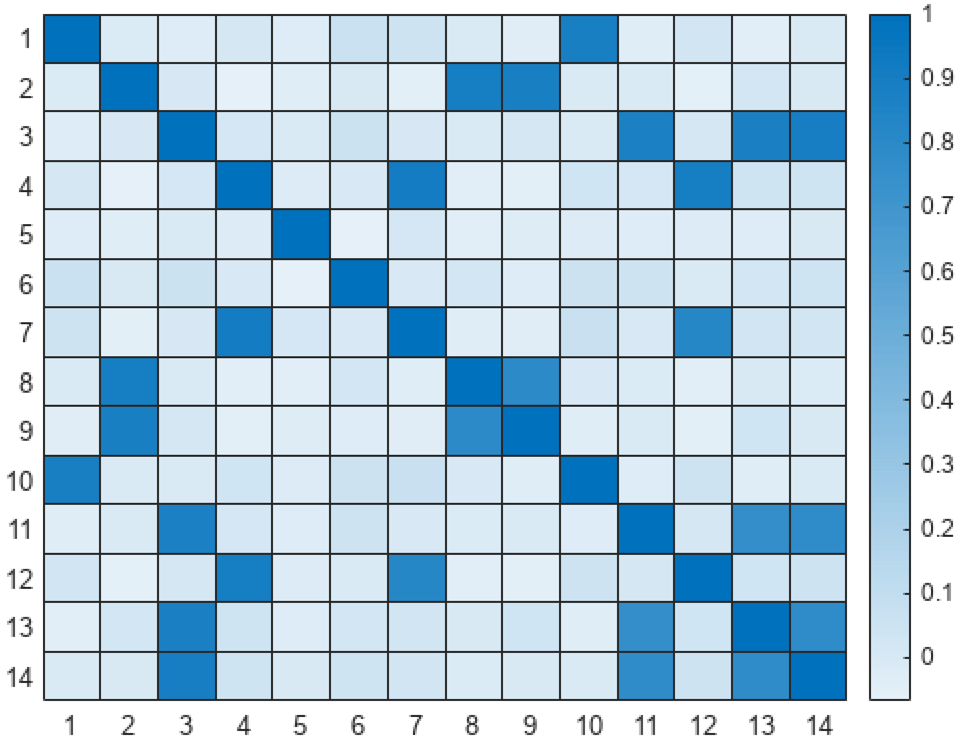
Create Hierarchical Risk Parity Portfolio

This example shows how to compute a hierarchical risk parity (HRP) portfolio. You can use HRP as a technique for portfolio diversification where the assets are divided and weighted according to a hierarchical tree structure. The weights of the assets within a cluster and between clusters can be assigned in many ways. A few ideas of the ways to allocate the weights are:

- Compute an inverse variance portfolio within each cluster. Then, allocate weights to each cluster using a value proportional to the inverse of the variance of the cluster's portfolio.
- Compute a risk parity portfolio within each cluster. Then, use a risk parity allocation strategy to assign each cluster's weights. The risk parity between clusters uses the covariance matrix between the cluster's portfolios. This example focuses on this allocation strategy.
- Use a bisection approach like the one described in Lopez de Prado [1 on page 4-296]. For more information, see the example Asset Allocation - Hierarchical Risk Parity.

Begin by loading the data and looking at the correlation between the assets returns.

```
% Load data
assetRetn = readmatrix("./retns_assets.txt");
[nSample,nAssets] = size(assetRetn);

% Compute covariance and correlation matrices
Sigma = cov(assetRetn);
C = corrcov(Sigma);
heatmap(C);
```



Hierachical Clustering

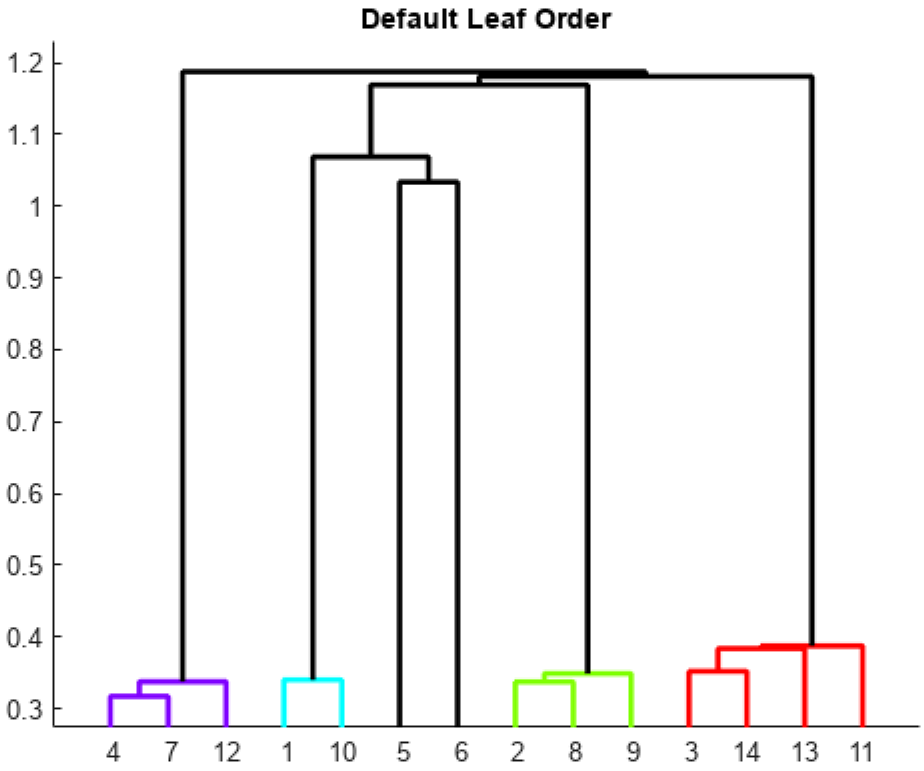
Hierarchical clustering is a common clustering technique in machine learning. In the context of asset allocation, a hierarchical clustering algorithm is applied to find the distance or similarity between each pair of assets and group them into a multilevel binary hierarchical tree.

Begin by deffining a measure of likeness or distance between the assets. The more correlated two assets are, the closer they should be.

```
% Compute the correlation distance matrix
distCorr = ((1-C)/2).^0.5;
```

Use the `linkage` function to compute the matrix that encodes the hierarchical tree of the assets in the universe. Them use the `dendrogram` function to visualize the hierarchical structure.

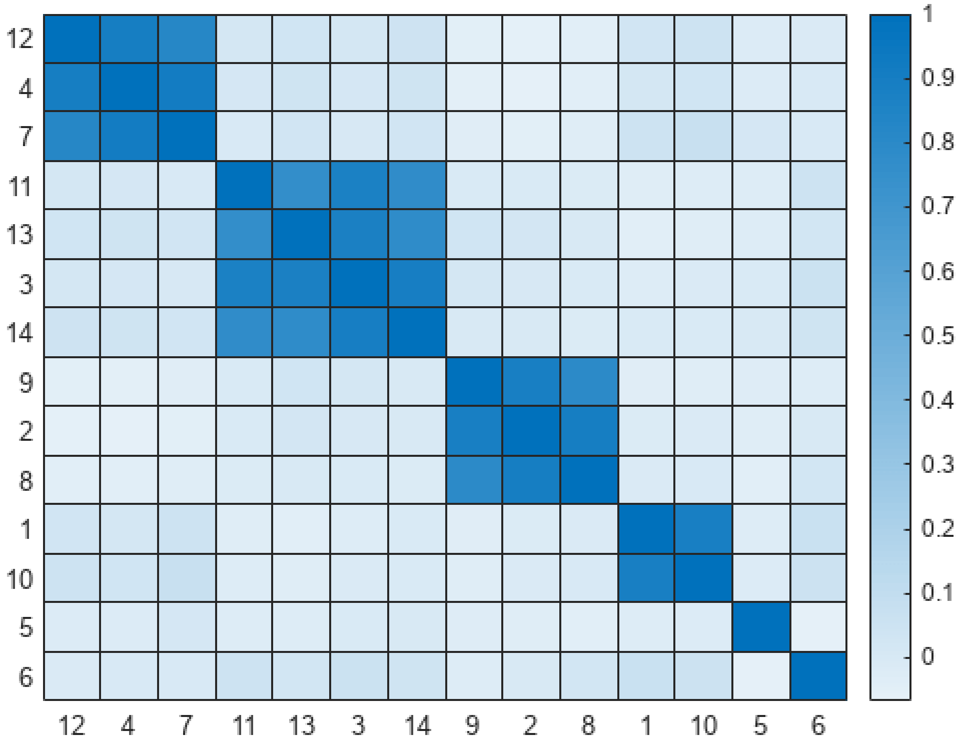
```
% Compute the linkage
link = linkage(distCorr);
figure;
h = dendrogram(link, ColorThreshold='default');
set(h, LineWidth=2);
title('Default Leaf Order');
```



The covariance or correlation matrix can be rearranged to be very close to a block diagonal matrix using the information obtained from the hierarchical tree. Each block in the diagonal shows the assets that are closely related.

```
% Sort assets for quasi-diagonalization
nLeafNodes = size(link,1) + 1;
```

```
rootNodeId = 2*nLeafNodes - 1;
sortedIdx = getLeafNodesInGroup(rootNodeId,link);
heatmap(C(sortedIdx,sortedIdx), XData=sortedIdx, YData=sortedIdx);
```



The plot shows that there are 6 blocks of assets that are closer together. You can divide the assets into 6 clusters using the `cluster` function.

```
% Get clusters
T = cluster(link, MaxClust=6);
```

Hierarchical Risk Parity Algorithm

Given a clustering of the assets, the HRP algorithm presented in this example follows these steps:

1. Build a risk parity portfolio within each cluster. The `hrpPortfolio` function in Local Functions on page 4-296 computes the HRP portfolio by receiving a vector with the cluster assignment and a covariance matrix Σ. Then, a risk parity portfolio is computed within each cluster by using `riskBudgetingPortfolio`. The `riskBudgetingPortfolio` function receives a reduced covariance matrix that only inlcudes the information of the assets within the cluster and it returns the weights of the assets in the cluster

$$w^j = \texttt{riskBudgetingPortfolio}(\Sigma^j),$$

where Σ^j is a matrix whose entries inlcude the covariance information only for the assets in the jth cluster.

2. Compute each cluster's weight usign the covariance between each cluster's portfolio. Now, the `riskBudgetingPortfolio` function receives a matrix (Γ) that represents the covariance between the cluster's portfolios (w^j).

$\Gamma_{\text{ij}} = W^T \Sigma W,$

where $W_{\text{ij}} = w_i^j$ if asset i is in cluster j, otherwise $W_{\text{ij}} = 0$. In other words $W = \left(w^1 | \ldots | w^K\right)$, for K clusters and $\gamma =$`riskBudgetingPortfolio(`Γ`).`

3. The final asset allocation is given by the cluster's risk parity portfolio (w^j) multiplied by each cluster's weight (γ_j)

$w = W\gamma.$

```
% Compute HRP portfolio
wHRP = hrpPortfolio(T,Sigma)
```

```
wHRP = 14×1

    0.0854
    0.0584
    0.0423
    0.0567
    0.1867
    0.1648
    0.0521
    0.0540
    0.0539
    0.0755
       ⋮
```

Compare HRP and Mean-Variance Portfolios

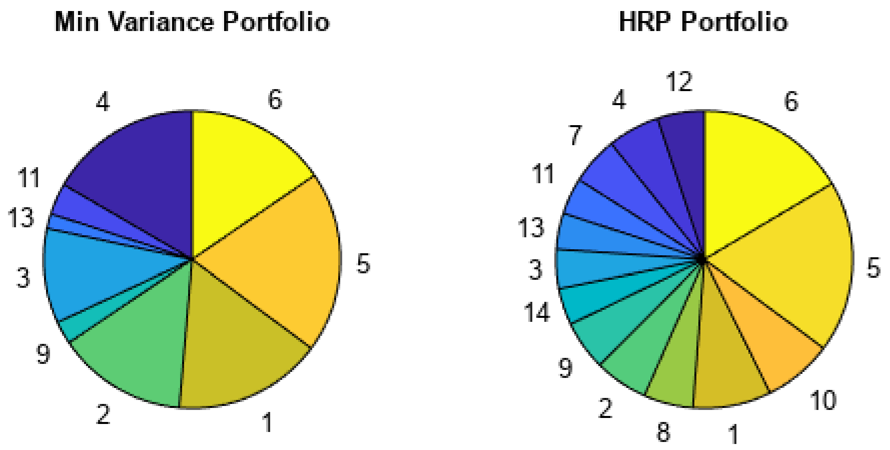
Define a long-only, fully-invested mean-variance `Portfolio` object. Then, compute the associated minimum variance portfolio.

```
% Define Portfolio object
p = Portfolio(AssetMean=mean(assetRetn), AssetCovar=Sigma);
p = setDefaultConstraints(p); % long-only, fully-invested portfolio
% Min variance portfolio
wMV = estimateFrontierLimits(p,'min');
```

Visualize the resulting allocations from these two strategies.

```
% Create pie chart labels to improve plot reading
labels = 1:nAssets;
labels = string(labels);

% Sort assets following quasi-diagonalization order
labels = labels(sortedIdx);
wMV = wMV(sortedIdx);
wHRP = wHRP(sortedIdx);

% Plot pie charts
```

```
tiledlayout(1,2);
% Min variance portfolio
nexttile
pie(wMV(wMV>=1e-8),labels(wMV>=1e-8))
title('Min Variance Portfolio',Position=[0,1.5]);
% HRP portfolio
nexttile
pie(wHRP,labels)
title('HRP Portfolio',Position=[0,1.5]);
```



You can see that the HRP portfolio is a much more diversified portfolio as compared to the portfolio obtained using the traditional mean-variance approach. In addition, you can see the following:

- The assets that were not correlated with others, assets 5 and 6, represent a much larger area of the pie. In fact, the sum of the areas of assets that are in the same cluster (for example, assets 1 and 10 or assets 11, 13, 3 and 14) are close to the individual areas of assets 5 and 6. This shows that the weights are divided somewhat evenly between clusters and that each cluster's weight is divided somewhat evenly among the assets within the cluster. This is consistent with the HRP theory.
- The risk parity portfolio of fully correlated assets with the same variance is an equally weighted portfolio. The same happens for assets that are completely uncorrelated. Since all the assets in the universe have a similar variance, and assets between clusters are almost uncorrelated, the weights allocated to each cluster are almost even. And, since the assets within a cluster are meant to be highly correlated, the weights of the assets within a culster are evenly distibuted.

References

1 Lopez de Prado, M. "Building Diversified Portfolios That Outperform Out of Sample." *The Journal of Portfolio Management*. *42*(4), 59-69, 2016.

Local Functions

```
function pwgt = hrpPortfolio(T,Sigma)
% Function that computes a hierarchical risk parity portfolio. The
% algorithm first computes a risk parity portfolio for each cluster. Then,
% each cluster is assigned a weight based on a risk parity allocation of
% the covariance between the cluster's portfolios.

% Get the problem information.
nAssets = size(Sigma,1);
nClusters = max(T);

% Compute the risk parity portfolio within each cluster.
W = zeros(nAssets,nClusters);
for i = 1:nClusters
    % Identify assets in cluster i and the sub-covariance matrix.
    idx = T == i;
    tempSigma = Sigma(idx,idx);
    % Compute the risk parity portfolio of cluster i.
    W(idx,i) = riskBudgetingPortfolio(tempSigma);
end

% Compute the covariance between the risk parity portfolios of each
% cluster.
covCluster = W'*Sigma*W;

% Compute the weights of each cluster.
wBetween = riskBudgetingPortfolio(covCluster);

% Multiply the weight assigned to each cluster with its portfolio and
% assign to the corresponding assets.
pwgt = W*wBetween;

end

function idxInGroup = getLeafNodesInGroup(nodeId, link)
% getLeafNodesInGroup finds all leaf nodes for a given node id
% in a linkage matrix.

nLeaves= size(link, 1)+1;
if nodeId > nLeaves
    tempNodeIds = link(nodeId-nLeaves,1:2);
    idxInGroup = [getLeafNodesInGroup(tempNodeIds(1), link), ...
        getLeafNodesInGroup(tempNodeIds(2), link)];
else
    idxInGroup = nodeId;
```

```
end
end
```

See Also

`portfolioRiskContribution` | `riskBudgetingPortfolio`

Related Examples

- “Risk Budgeting Portfolio” on page 4-280
- “Backtest Using Risk-Based Equity Indexation” on page 4-285

External Websites

- Asset Allocation - Hierarchical Risk Parity (2 min 42 sec)

Backtest Strategies Using Deep Learning

Construct trading strategies using a deep learning model and then backtest the strategies using the Financial Toolbox™ backtesting framework. The example uses Deep Learning Toolbox™ to train a predictive model from a set of time series and demonstrates the steps necessary to convert the model output into trading signals. It builds a variety of trading strategies that backtest the signal data over a 5-year period.

This example illustrates the following workflow:

1. Load price data for a set of energy commodities on page 4-298.
2. Clean and trim data on page 4-299.
3. Use the historical data to train a long short-term memory (LSTM) network to predict the change in energy prices over the next trading day on page 4-300.
4. Use the LSTM network to build a timetable of trading signal data for the backtest engine on page 4-305.
5. Construct trading strategies that allocate capital based on the trading signals on page 4-306.
6. Backtest the strategies using the backtesting framework on page 4-306.
7. Examine the backtest results on page 4-307.

The focus of this example is on the workflow from data, to a trained model, to trading strategies, and finally to a backtest of the strategies. The deep learning model, its output, the subsequent trading signals, and the strategies are fictional. The intent is only to show the steps for developing and deploying this type of model.

Load Data

Load the historical price data. This data set contains daily spot prices for 12 different energy products ranging from 1986 to 2021 and consists of the following time series:

- WTI — West Texas Intermediate light crude oil
- Brent — Brent light crude oil
- NaturalGas — Henry Hub natural gas
- Propane — Mon Belvieu propane
- Kerosene — US Gulf Coast kerosene-type jet fuel
- HeatingOil — New York Harbor no. 2 heating oil
- GulfRegular — US Gulf Coast conventional gasoline
- LARegular — Los Angeles reformulated RBOB regular gasoline
- NYRegular — New York Harbor conventional gasoline
- GulfDiesel — US Gulf Coast ultra-low sulfur no. 2 diesel
- LADiesel — Los Angeles ultra-low sulfur CARB diesel
- NYDiesel — New York Harbor ultra-low sulfur no. 2 diesel

The source of this data is the US Energy Information Administration (Nov 2021).

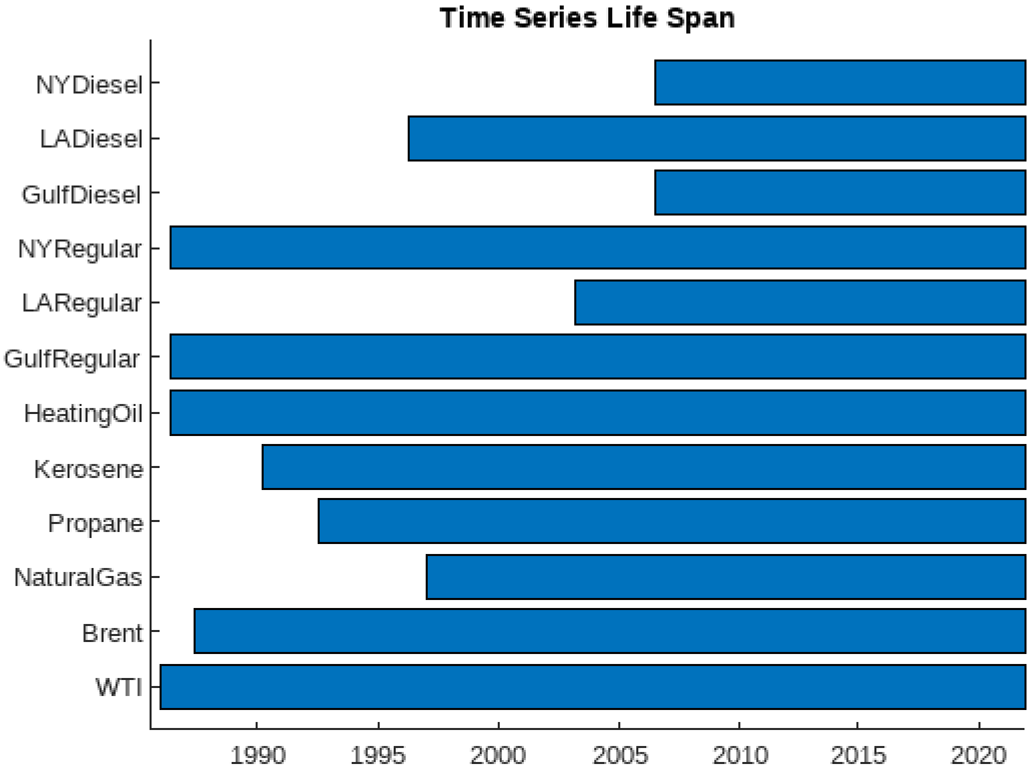
```
priceData = load('energyPrices.mat','energyPrices');
priceData = priceData.energyPrices;
tail(priceData)
```

```
        Time          WTI      Brent    NaturalGas    Propane    Kerosene    HeatingOil    GulfRegu
    ___________    _____    _____    __________    _______    ________    __________    ________

    22-Oct-2021    84.53    85.43         5.1       1.485      2.312        2.414         2.481
    25-Oct-2021    84.64    84.85        5.72       1.378      2.326        2.429         2.506
    26-Oct-2021    85.64    85.11        5.59       1.398      2.339        2.436         2.552
    27-Oct-2021    82.66    84.12        5.91       1.365      2.271        2.368         2.469
    28-Oct-2021    82.78     83.4        5.68        1.36      2.278        2.363         2.471
    29-Oct-2021     83.5     83.1        5.49       1.383      2.285        2.342         2.485
    01-Nov-2021    84.08    84.51        5.22       1.385      2.301        2.364         2.457
    02-Nov-2021    83.91    84.42        5.33       1.388        2.3        2.405         2.466
```

Clean and Trim Data

The price datasets do not all start at the same time. Some datasets start later than others and have fewer data points. The following plot shows the time span for each price series.

```
seriesLifespanPlot(priceData)
```



To avoid large spans of missing data, remove the series with shorter histories.

```
priceData = removevars(priceData,["NYDiesel","GulfDiesel","LARegular"]);
```

The remaining table variables contain sporadic missing elements (NaNs) due to holidays or other reasons. Missing data is handled in a variety of ways depending on the dataset. In some cases, it may be appropriate to interpolate or use the `fillmissing` function. In this example, you can remove the remaining NaN prices.

```
priceData = rmmissing(priceData);
```

Then, convert the price data to a return series using the `tick2ret` function. The final dataset consists of nine price series with daily data from 1997 through 2021.

```
returnData = tick2ret(priceData)
```

```
returnData=6167×9 timetable
       Time           WTI           Brent       NaturalGas     Propane       Kerosene     Heating
    ___________    __________    ___________    __________    __________    __________    ______

    08-Jan-1997      0.011429     0.00080775    -0.0052356             0      0.012931      0.010
    09-Jan-1997    -0.0094162      0.0020178         -0.05     -0.036969    -0.0085106    -0.0013
    10-Jan-1997    -0.0057034      -0.024567      0.085873     0.0095969     -0.010014     -0.012
    13-Jan-1997     -0.036329      -0.033443      0.020408     -0.024715     -0.034682     -0.037
    15-Jan-1997      0.029762     -0.0042717         0.085     -0.048733      0.023952      0.021
    16-Jan-1997     -0.019268              0      0.085253     -0.028689     -0.019006     -0.020
    17-Jan-1997    -0.0019646      -0.018876      -0.16985     -0.016878     -0.020864         -0
    20-Jan-1997     -0.011811    -0.00043725      -0.16624     -0.027897     -0.022831     -0.021
    21-Jan-1997     -0.011952      0.0052493     -0.082822     -0.004415     -0.014019     -0.020
    22-Jan-1997     -0.016129     -0.0021758      0.020067    -0.0044346      0.031596      0.019
    23-Jan-1997     -0.022541              0     -0.029508    -0.0022272    -0.0061256      -0.01
    24-Jan-1997             0     -0.0056694      -0.11486     -0.075893      0.010786     0.0060
    27-Jan-1997             0      -0.010526        0.1374      0.016908      0.012195      0.009
    28-Jan-1997     0.0020964      0.0026596       0.02349    -0.0047506     0.0090361    -0.0089
    29-Jan-1997      0.025105       0.017241     -0.045902     -0.042959      0.059701      0.033
    30-Jan-1997      0.012245       0.018253     -0.017182             0      0.016901      0.023
      ⋮
```

Prepare Data for Training LSTM Model

Prepare and partition the dataset in order to train the LSTM model. The model uses a 30-day rolling window of trailing feature data and predicts the next day price changes for four of the assets: Brent crude oil, natural gas, propane, and kerosene.

```
% Model is trained using a 30-day rolling window to predict 1 day in the
% future.
historySize = 30;
futureSize = 1;

% Model predicts returns for oil, natural gas, propane, and kerosene.
outputVarName = ["Brent" "NaturalGas", "Propane" "Kerosene"];
numOutputs = numel(outputVarName);

% start_idx and end_idx are the index positions in the returnData
% timetable corresponding to the first and last date for making a prediction.
start_idx = historySize + 1;
end_idx   = height(returnData) - futureSize + 1;
numSamples = end_idx - start_idx + 1;

% The date_vector variable stores the dates for making predictions.
date_vector = returnData.Time(start_idx-1:end_idx-1);
```

Convert the `returnData` timetable to a `numSamples`-by-`1` cell array. Each cell contains a `numFeatures`-by-`seqLength` matrix. The response variable is a `numSamples`-by-`numResponses` matrix.

```
network_features  = cell(numSamples,1);
network_responses = zeros(numSamples,numOutputs);

for j = 1:numSamples
    network_features{j} = (returnData(j:j+historySize-1,:).Variables)';
    network_responses(j,:) = ...
        (returnData(j+historySize:j+historySize+futureSize-1,outputVarName).Variables)';
end
```

Split the `network_features` and the `network_responses` into three parts: training, validation, and backtesting. Select the backtesting set as a set of sequential data points. The remainder of the data is randomly split into a training and a validation set. Use the validation set to prevent overfitting while training the model. The backtesting set is not used in the training process, but it is reserved for the final strategy backtest.

```
% Specify rows to use in the backtest (31-Dec-2015 to 2-Nov-2021).
backtest_start_idx = find(date_vector < datetime(2016,1,1),1,'last');
backtest_indices = backtest_start_idx:size(network_responses,1);

% Specify data reserved for the backtest.
Xbacktest = network_features(backtest_indices);
Tbacktest = network_responses(backtest_indices,:);

% Remove the backtest data.
network_features = network_features(1:backtest_indices(1)-1);
network_responses = network_responses(1:backtest_indices(1)-1,:);

% Partition the remaining data into training and validation sets.
rng('default');
cv_partition = cvpartition(size(network_features,1),'HoldOut',0.2);

% Training set
Xtraining = network_features(~cv_partition.test,:);
Ttraining = network_responses(~cv_partition.test,:);

% Validation set
Xvalidation = network_features(cv_partition.test,:);
Tvalidation = network_responses(cv_partition.test,:);
```

Define LSTM Network Architecture

Specify the network architecture as a series of layers. For more information on LSTM networks, see “Long Short-Term Memory Neural Networks” (Deep Learning Toolbox). The Deep Network Designer (Deep Learning Toolbox) is a powerful tool for designing deep learning models.

```
numFeatures = width(returnData);
numHiddenUnits_LSTM = 10;

layers_LSTM = [ ...
    sequenceInputLayer(numFeatures)
    lstmLayer(numHiddenUnits_LSTM)
    layerNormalizationLayer
    lstmLayer(numHiddenUnits_LSTM)
    layerNormalizationLayer
    lstmLayer(numHiddenUnits_LSTM,'OutputMode','last')
    layerNormalizationLayer
    fullyConnectedLayer(numOutputs)
    regressionLayer];
```

Specify Training Options for LSTM Model

Next, you specify training options using the `trainingOptions` (Deep Learning Toolbox) function. Many training options are available and their use varies depending on your use case. Use the Experiment Manager (Deep Learning Toolbox) to explore different network architectures and sets of network hyperparamters.

```
max_epochs = 500;
mini_batch_size = 128;
learning_rate = 1e-4;

options_LSTM = trainingOptions('adam', ...
    'Plots','training-progress', ...
    'Verbose',0, ...
    'MaxEpochs',max_epochs, ...
    'MiniBatchSize',mini_batch_size, ...
    'Shuffle','every-epoch', ...
    'ValidationData',{Xvalidation,Tvalidation}, ...
    'ValidationFrequency',50, ...
    'ValidationPatience',10, ...
    'InitialLearnRate',learning_rate, ...
    'GradientThreshold',1);
```

Train LSTM Model

Train the LSTM network. Use the `trainNetwork` (Deep Learning Toolbox) function to train the network until the network meets a stopping criteria. This process can take several minutes depending on the computer running the example. For more information on increasing the network training performance, see “Scale Up Deep Learning in Parallel, on GPUs, and in the Cloud” (Deep Learning Toolbox).

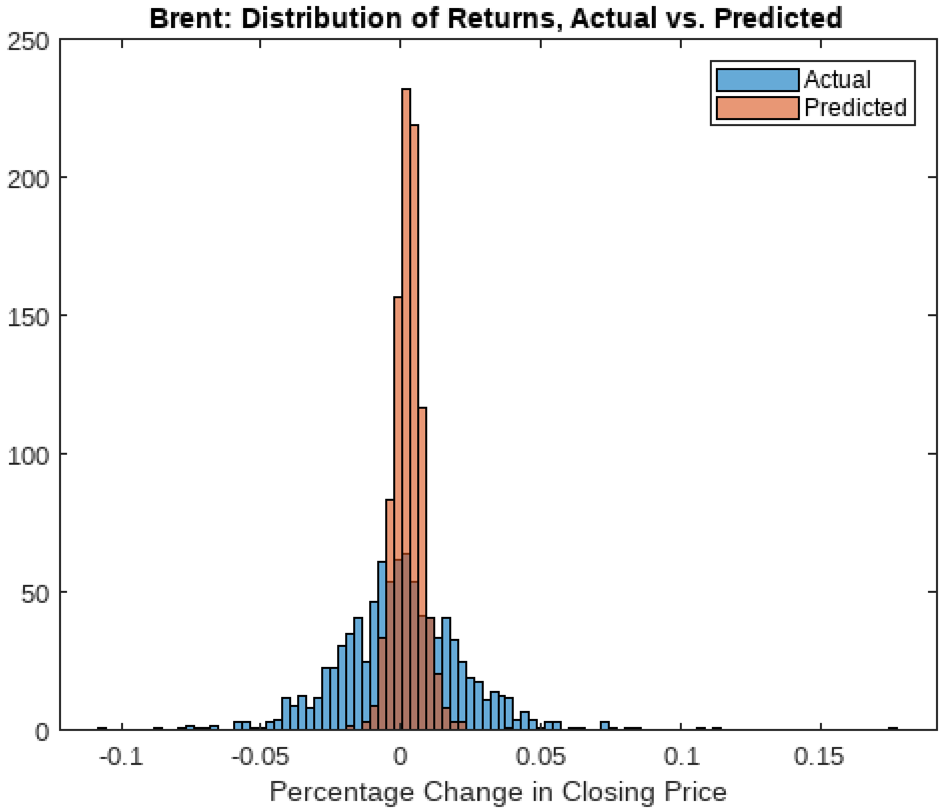
To avoid waiting for the network training, load the pretrained network by setting the `doTrain` flag to `false`. To train the network using `trainNetwork` (Deep Learning Toolbox), set the `doTrain` flag to `true`.

```
doTrain = false;

if doTrain
    % Train the LSTM network.
    net_LSTM = trainNetwork(Xtraining,Ttraining,layers_LSTM,options_LSTM);
else
    % Load the pretrained network.
    load lstmBacktestNetwork
end
```

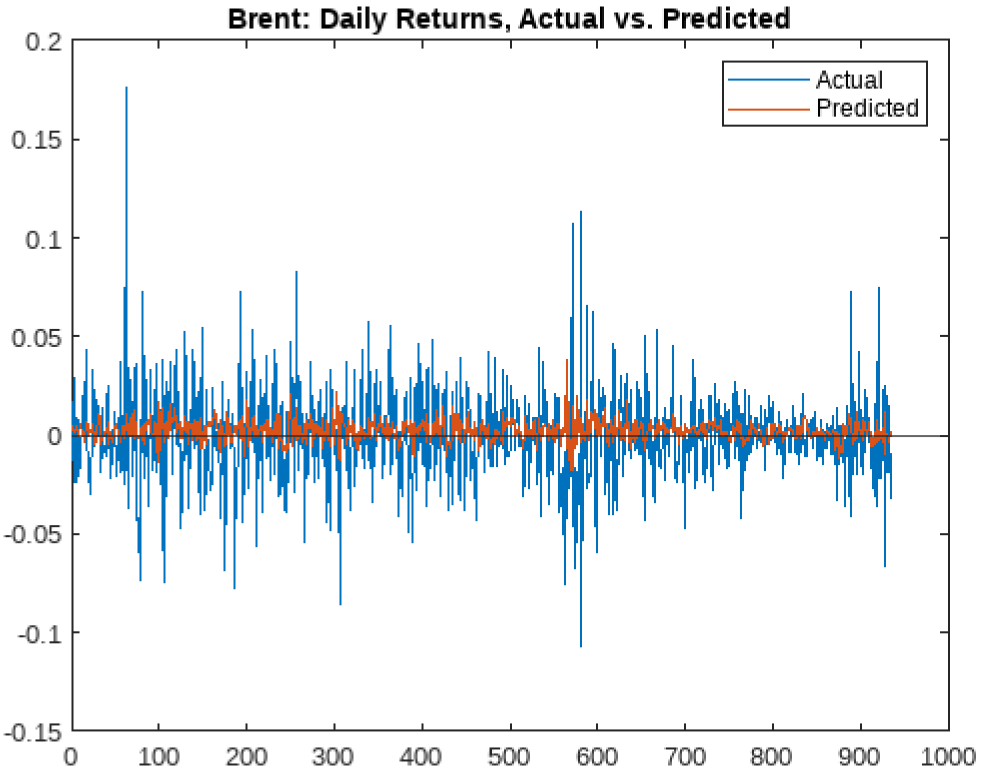
Visualize Training Results

Visualize the results of the trained model by comparing the predicted values against the actual values from the validation set.

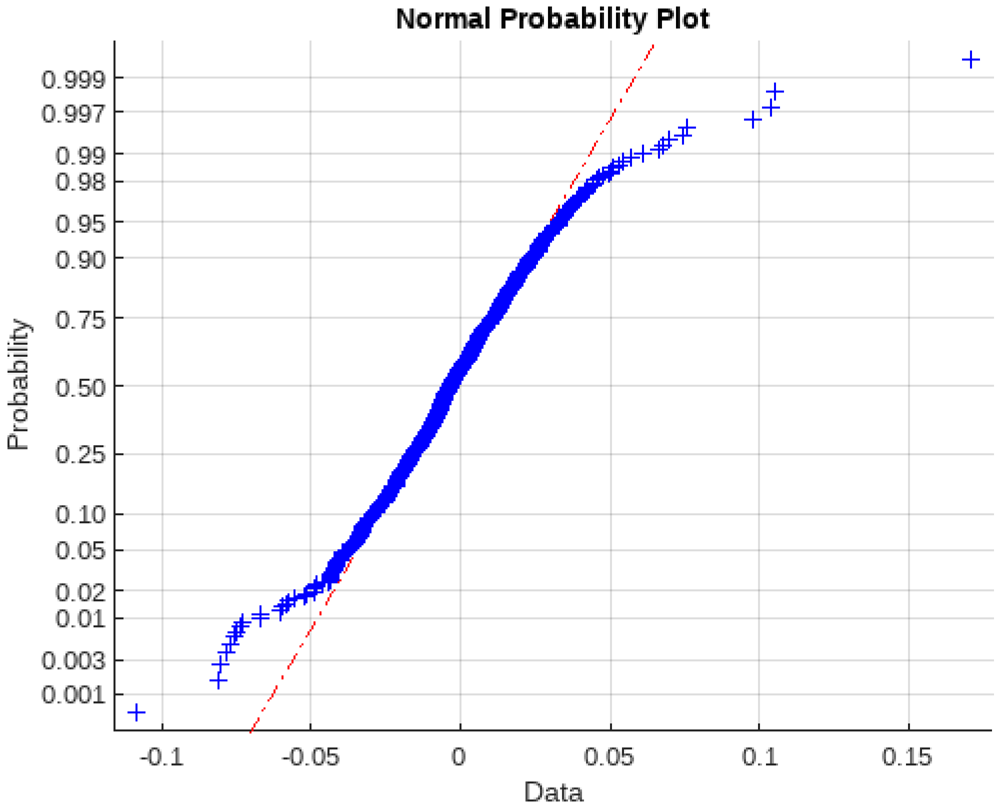
```
% Compare the actual returns to model predicted returns.
actual    = Tvalidation;
predicted = predict(net_LSTM,Xvalidation,'MiniBatchSize',mini_batch_size);

% Overlay histogram of actual vs. predicted returns for the validation set.
output_idx = Oil ▾;
figure;

[~,edges] = histcounts(actual(:,output_idx),100);
histogram(actual(:,output_idx),edges);
hold on
histogram(predicted(:,output_idx),edges)
hold off
xlabel('Percentage Change in Closing Price')
legend('Actual','Predicted')
title(sprintf('%s: Distribution of Returns, Actual vs. Predicted', outputVarName(output_idx)))
```



```
% Display the predicted vs. actual daily returns for the validation set.
figure
plot(actual(:,output_idx))
hold on
plot(predicted(:,output_idx))
yline(0)
legend({'Actual','Predicted'})
title(sprintf('%s: Daily Returns, Actual vs. Predicted', outputVarName(output_idx)))
```



```
% Examine the residuals.
residuals = actual(:,output_idx) - predicted(:,output_idx);
figure;
normplot(residuals);
```



The actual data has fatter tails than the trained model predictions. The model predictions are not accurate, but the goal of this example is to show the workflow from loading data, to model development, to backtesting. A more sophisticated model with a larger and more varied set of training data is likely to have more predictive power.

Prepare Backtest Data

Use the predictions from the LSTM model to build the backtest strategies. You can post-process the model output in a number of ways to create trading signals. However, for this example, take the model regression output and convert it to a timetable.

Use `predict` (Deep Learning Toolbox) with the trained network to generate model predictions over the backtest period.

```
backtestPred_LSTM = predict(net_LSTM,Xbacktest,'MiniBatchSize',mini_batch_size);
```

Convert the predictions to a trading signal timetable.

```
backtestSignalTT = timetable(date_vector(backtest_indices),backtestPred_LSTM);
```

Construct the prices timetable corresponding to the backtest time span. The backtest trades in and out of the four energy commodities. The prices timetable has the closing price for the day on which the prediction is made.

```
backtestPriceTT = priceData(date_vector(backtest_indices),outputVarName);
```

Set the risk-free rate to be 1% annualized. The backtest engine also supports setting the risk-free rate to a timetable containing the historical daily rates.

```
risk_free_rate = 0.01;
```

Create Backtest Strategies

Use `backtestStrategy` to create four trading strategies based on the signal indicators. The following trading strategies are intended as examples to show how to convert the trading signals into actionable asset allocation strategies that you can then backtest:

- Long Only — Invest all capital across the assets with positive predicted return, proportional to their signal strength (predicted return).
- Long Short — Invest capital across the assets, both long and short positions, proportional to their signal strength.
- Best Bet — Invest all capital into the single asset with the highest predicted return.
- Equal Weight — Rebalance each day to equal-weighted allocation.

```
% Specify 10 basis points as the trading cost.
tradingCosts = 0.001;

% Invest in long positions proportionally to their predicted return.
LongStrategy = backtestStrategy('LongOnly',@LongOnlyRebalanceFcn, ...
    'TransactionCosts',tradingCosts, ...
    'LookbackWindow',1);

% Invest in both long and short positions proportionally to their predicted returns.
LongShortStrategy = backtestStrategy('LongShort',@LongShortRebalanceFcn, ...
    'TransactionCosts',tradingCosts, ...
    'LookbackWindow',1);

% Invest 100% of capital into single asset with highest predicted returns.
BestBetStrategy = backtestStrategy('BestBet',@BestBetRebalanceFcn, ...
    'TransactionCosts',tradingCosts, ...
    'LookbackWindow',1);

% For comparison, invest in an equal-weighted (buy low and sell high) strategy.
equalWeightFcn = @(current_weights,prices,signal) ones(size(current_weights)) / numel(current_we
EqualWeightStrategy = backtestStrategy('EqualWeight',equalWeightFcn, ...
    'TransactionCosts',tradingCosts, ...
    'LookbackWindow',0);
```

Put the `strategies` into an array and then use `backtestEngine` to create the backtesting engine.

```
strategies = [LongStrategy LongShortStrategy BestBetStrategy EqualWeightStrategy];

bt = backtestEngine(strategies,'RiskFreeRate',risk_free_rate);
```

Run Backtest

Use `runBacktest` to backtest the strategies over the backtest range.

```
bt = runBacktest(bt,backtestPriceTT,backtestSignalTT)
```

```
bt = 
  backtestEngine with properties:
```

```
              Strategies: [1×4 backtestStrategy]
            RiskFreeRate: 0.0100
          CashBorrowRate: 0
         RatesConvention: "Annualized"
                   Basis: 0
   InitialPortfolioValue: 10000
          DateAdjustment: "Previous"
               NumAssets: 4
                 Returns: [1462×4 timetable]
               Positions: [1×1 struct]
                Turnover: [1462×4 timetable]
                 BuyCost: [1462×4 timetable]
                SellCost: [1462×4 timetable]
```

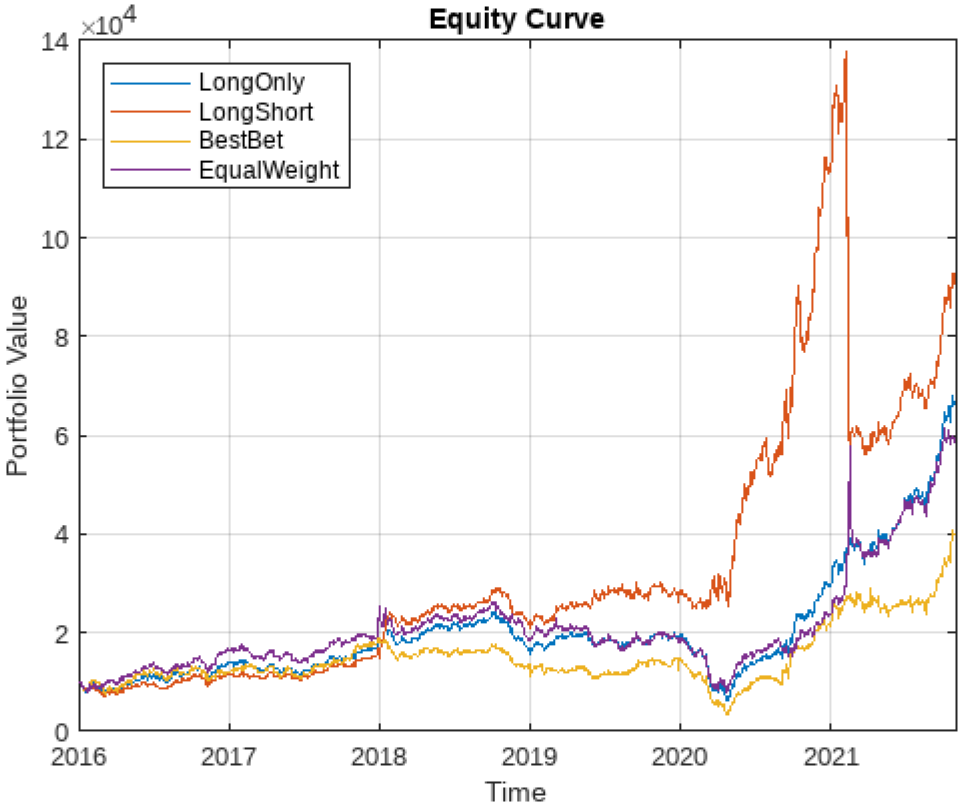
Examine Backtest Results

Use the `summary` and `equityCurve` functions to summarize and plot the backtest results. This model and its derivative trading strategies are not expected to be profitable in a realistic trading scenario. However, this example illustrates a workflow that should be useful for practitioners with more comprehensive data sets and more sophisticated models and strategies.

```
summary(bt)
```

```
ans=9×4 table
                       LongOnly     LongShort    BestBet     EqualWeight
                       _________    _________    ________    ___________

    TotalReturn           5.6962       8.3314      3.0248         4.8347
    SharpeRatio         0.062549     0.071321    0.044571       0.056775
    Volatility          0.025296     0.025795    0.031625       0.026712
    AverageTurnover       0.1828      0.22754      0.2459      0.0095931
    MaxTurnover          0.96059      0.97368           1            0.5
    AverageReturn      0.0016216     0.001879    0.001449       0.001556
    MaxDrawdown          0.73831      0.62935     0.81738        0.70509
    AverageBuyCost        3.6293       7.1838      3.8139        0.20262
    AverageSellCost       3.6225       7.2171      3.8071        0.19578
```

```
figure;
equityCurve(bt)
```



Local Functions

```
function new_weights = LongOnlyRebalanceFcn(current_weights,pricesTT,signalTT) %#ok<INUSD>
% Long only strategy, in proportion to the signal.

signal = signalTT.backtestPred_LSTM(end,:);

if any(0 < signal)
    signal(signal < 0) = 0;
    new_weights = signal / sum(signal);
else
    new_weights = zeros(size(current_weights));
end

end


function new_weights = LongShortRebalanceFcn(current_weights,pricesTT,signalTT) %#ok<INUSD>
% Long/Short strategy, in proportion to the signal

signal = signalTT.backtestPred_LSTM(end,:);
abssum = sum(abs(signal));

if 0 < abssum
    new_weights = signal / abssum;
else
    new_weights = zeros(size(current_weights));
```

```
end

end


function new_weights = BestBetRebalanceFcn(current_weights,pricesTT,signalTT) %#ok<INUSD>
% Best bet strategy, invest in the asset with the most upside.

signal = signalTT.backtestPred_LSTM(end,:);
new_weights = zeros(size(current_weights));
new_weights(signal == max(signal)) = 1;

end


function seriesLifespanPlot(priceData)
% Plot the lifespan of each time series.

% Specify all time series end on same day.
d2 = numel(priceData.Time);

% Plot the lifespan patch for each series.
numSeries = size(priceData,2);
for i = 1:numSeries
    % Find start date index.
    d1 = find(~isnan(priceData{:,i}),1,'first');
    % Plot patch.
    x = [d1 d1 d2 d2];
    y = i + [-0.4 0.4 0.4 -0.4];
    patch(x,y,[0 0.4470 0.7410])

    hold on
end
hold off

% Set the plot properties.
xlim([-100 d2]);
ylim([0.2 numSeries + 0.8]);

yticks(1:numSeries);
yticklabels(priceData.Properties.VariableNames');
flipud(gca);

years = 1990:5:2021;
xtick_idx = zeros(size(years));
for yidx = 1:numel(years)
    xtick_idx(yidx) = find(years(yidx) == year(priceData.Time),1,'first');
end
xticks(xtick_idx);
xticklabels(string(years));

title('Time Series Life Span');
```

```
end
```

See Also

Deep Network Designer | `trainNetwork` | `trainingOptions` | `backtestStrategy` | `backtestEngine` | `runBacktest` | `equityCurve` | `summary`

Related Examples

- “Backtest Investment Strategies Using Financial Toolbox™” on page 4-231
- “Backtest Investment Strategies with Trading Signals” on page 4-244
- “Backtest Using Risk-Based Equity Indexation” on page 4-285
- “Backtest with Brinson Attribution to Evaluate Portfolio Performance” on page 4-311
- “Backtest with Brinson Attribution to Evaluate Portfolio Performance” on page 4-311

Backtest with Brinson Attribution to Evaluate Portfolio Performance

This example shows how to compute Brinson attribution using the output of the MATLAB® backtest framework. The backtest framework allows you to build custom trading strategies and then backtest them against historical or simulated market data. The example uses the Brinson attribution function (`brinsonAttribution`) to explain a portfolio's performance versus a benchmark.

Load Data

Brinson attribution requires that a category is assigned to each asset. Load the `dowPortfolio.xlsx` data set, then assign each of the 30 stocks to a category.

```
% Read a table of daily adjusted close prices for 2006 DJIA stocks.
pricesTT = readtimetable('dowPortfolio.xlsx');

% Remove the index from the dataset.
pricesTT= removevars(pricesTT,'DJI');
num_assets = width(pricesTT);

% Set the sectors for each stock.
asset_categories = categorical(["Materials";"Financials";"Financials"; ...
    "Industrials";"Financials";"Industrials";"Materials"; ...
    "Communication Services";"Industrials";"Consumer Discretionary"; ...
    "Consumer Discretionary";"Industrials";"Information Technology"; ...
    "Information Technology";"Information Technology";"Health Care"; ...
    "Financials";"Consumer Staples";"Consumer Discretionary"; ...
    "Industrials";"Consumer Staples";"Health Care"; ...
    "Information Technology";"Health Care";"Consumer Staples"; ...
    "Communication Services";"Industrials";"Communication Services"; ...
    "Consumer Staples";"Energy"]);

% Display the stock categories.
Symbol = pricesTT.Properties.VariableNames(:);
table(Symbol,asset_categories)
```

```
ans=30×2 table
     Symbol         asset_categories
    ________    ______________________

    {'AA'  }    Materials
    {'AIG' }    Financials
    {'AXP' }    Financials
    {'BA'  }    Industrials
    {'C'   }    Financials
    {'CAT' }    Industrials
    {'DD'  }    Materials
    {'DIS' }    Communication Services
    {'GE'  }    Industrials
    {'GM'  }    Consumer Discretionary
    {'HD'  }    Consumer Discretionary
    {'HON' }    Industrials
    {'HPQ' }    Information Technology
    {'IBM' }    Information Technology
    {'INTC'}    Information Technology
    {'JNJ' }    Health Care
      ⋮
```

Define and Create Backtest Strategies

Multiperiod Brinson attribution expects fixed, regular time periods. In this case, the backtest strategy rebalance schedules will match the Brinson periods. You can set the rebalance dates to be monthly starting from January first.

```
% Rebalance on the first trading day of each month.
rebalance_schedule = datetime(2006,1,1):calmonths(1):datetime(2006,12,1);
for idx = 1:numel(rebalance_schedule)
    priceIdx = find(rebalance_schedule(idx) <= pricesTT.Dates,1,'first');
    rebalance_schedule(idx) = pricesTT.Dates(priceIdx);
end
```

The risk budgeting strategy needs historical data in order to set initial weights. You can use the first month of price data as a warm-up period to set the initial weights. The backtest begins at the end of the warm-up period.

```
% Use first month as warm-up period for the risk budgeting strategy.
start_date = rebalance_schedule(2);

% Get the first month of prices to initialize the risk budgeting strategy.
initial_prices = pricesTT(pricesTT.Dates <= start_date,:);
```

Brinson attribution measures portfolio manager performance against some benchmark. For this example, use a simple equal-weighted strategy as the benchmark. This example examines the performance of the following two strategies relative to this benchmark:

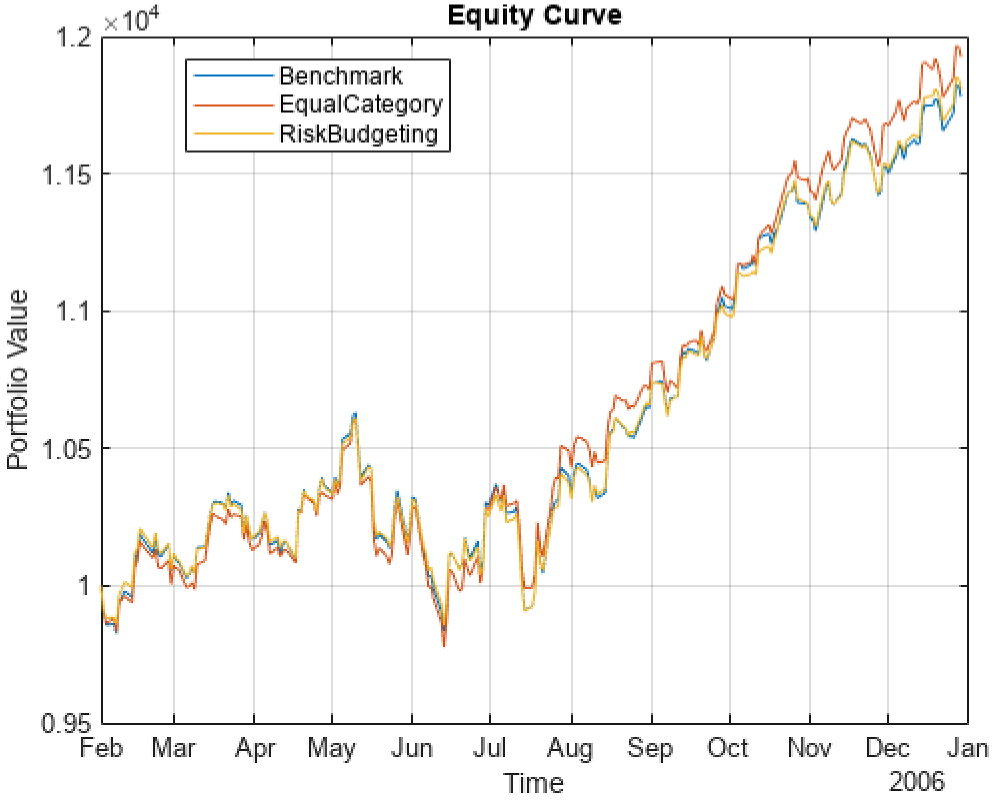
- Equal Category — This strategy allocates resources equally *per category*, and then allocates equal weights for stocks within each category.
- Risk Budgeting — This strategy sets weights for stocks with the goal of minimizing each asset's risk contribution.

Create a `backtestStrategy` object for the benchmark and the two candidate strategies. Define each of these strategies to use the same rebalance schedule and assign the appropriate initial weights.

```
% Define the Benchmark strategy.
benchmark = backtestStrategy('Benchmark',@equalWeightRebalanceFcn, ...
    'RebalanceFrequency',rebalance_schedule, ...
    'InitialWeights',computeEqualWeights(num_assets), ...
    'UserData',struct());

% Define the equal category strategy.
equalCategory = backtestStrategy('EqualCategory',@equalCategoryRebalanceFcn, ...
    'RebalanceFrequency',rebalance_schedule, ...
    'InitialWeights',computeEqualCategory(asset_categories), ...
    'UserData',struct('Categories',asset_categories));

% Define the risk budgeting strategy.
riskBudgeting = backtestStrategy('RiskBudgeting',@riskBudgetingRebalanceFcn, ...
    'RebalanceFrequency',rebalance_schedule, ...
    'InitialWeights',computeRiskBudgeting(initial_prices));
```

Use backtestEngine to create a backtestEngine object for the strategies and then use runBacktest to run the backtest. Display the equity curve using equityCurve and then use summary to display the backtest summary table. Both the equal category and risk budgeting strategies outperform the benchmark.

```
strategies = [benchmark,equalCategory,riskBudgeting];
bt = backtestEngine(strategies);
bt = runBacktest(bt,pricesTT,'Start',start_date);
equityCurve(bt)
```



```
summary(bt)
```

```
ans=9×3 table
                         Benchmark      EqualCategory    RiskBudgeting
                         __________     _____________    _____________

    TotalReturn             0.17811          0.19249          0.18065
    SharpeRatio             0.11671          0.12645          0.12393
    Volatility            0.0062905        0.0062211        0.0059849
    AverageTurnover      0.00073556       0.00076049        0.0015965
    MaxTurnover            0.025681         0.028611         0.080538
    AverageReturn        0.00073259       0.00078493       0.00074008
    MaxDrawdown             0.07502         0.078323         0.071364
    AverageBuyCost                0                0                0
    AverageSellCost               0                0                0
```

Prepare Backtest Results for Brinson Attribution

Set the strategy name of the benchmark.

```
benchmark_name = 'Benchmark';
```

Define Brinson periods by the rebalance dates and the backtest end date.

```
period_dates = [rebalance_schedule(2:end), pricesTT.Dates(end)];
```

Compute Brinson Attribution for Backtested Investment Strategy

Select whether to analyze the equal category or risk budgeting strategy. Then, use the `backtest2brinson` function in Local Functions on page 4-317 to compute `asset_table` that is formatted for use with the `brinsonAttribution` function. For more information on the format of the `asset_table`, see the description for the `AssetTable` input argument in the `brinsonAttribution` function.

```
% Select strategy to analyze
strategy_name = EqualCategory ▾;

asset_table = backtest2brinson(bt,strategy_name,benchmark_name,pricesTT,asset_categories,period_
head(asset_table)
```

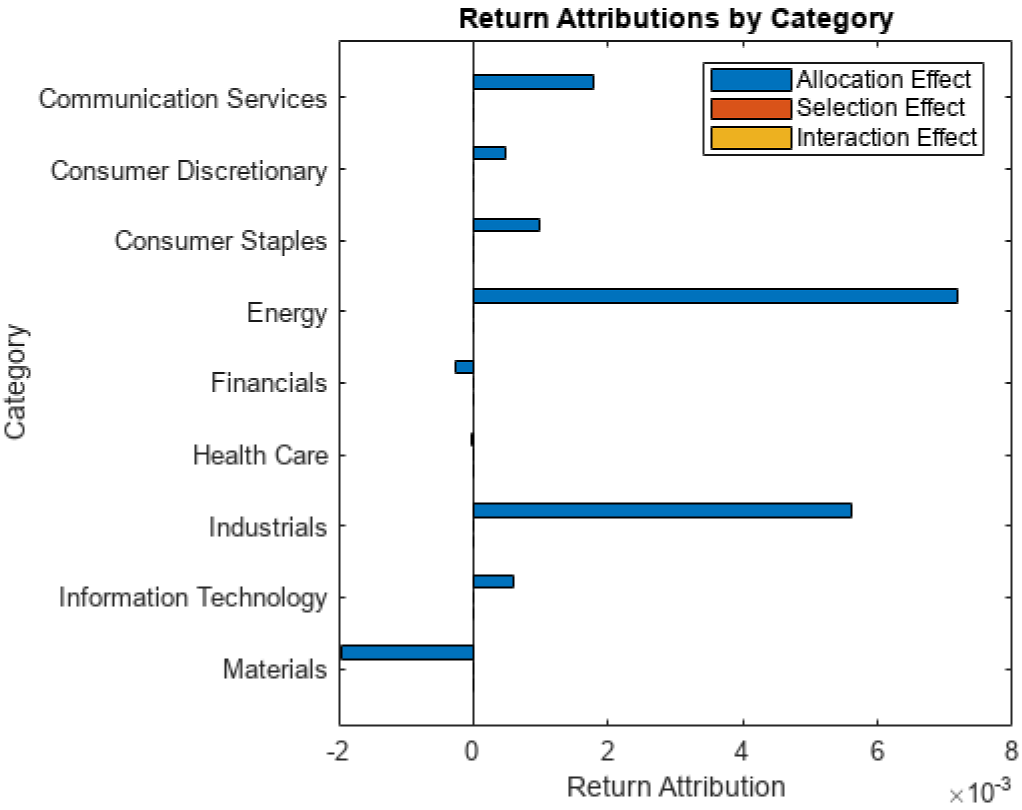
```
    Period     Name       Return              Category              PortfolioWeight    BenchmarkWeight
    ______    _____    _________    ______________________    _______________    _______________

      1       "AA"     -0.050283    Materials                       0.055556           0.033333
      1       "AIG"    0.0015356    Financials                      0.027778           0.033333
      1       "AXP"     0.028439    Financials                      0.027778           0.033333
      1       "BA"      0.022929    Industrials                     0.018519           0.033333
      1       "C"       0.015045    Financials                      0.027778           0.033333
      1       "CAT"     0.074416    Industrials                     0.018519           0.033333
      1       "DD"      0.037791    Materials                       0.055556           0.033333
      1       "DIS"      0.11182    Communication Services          0.037037           0.033333
```

```
% Compute Brinson attribution
brinson = brinsonAttribution(asset_table);
summary(brinson)
```

```
ans=11×1 table
                                      Brinson Attribution Summary
                                      ___________________________

    Total Number of Assets                          30
    Number of Assets in Portfolio                   30
    Number of Assets in Benchmark                   30
    Number of Periods                               11
    Number of Categories                             9
    Portfolio Return                           0.19249
    Benchmark Return                           0.17811
    Active Return                             0.014376
    Allocation Effect                         0.014376
    Selection Effect                        8.5567e-18
    Interaction Effect                     -1.1701e-19
```

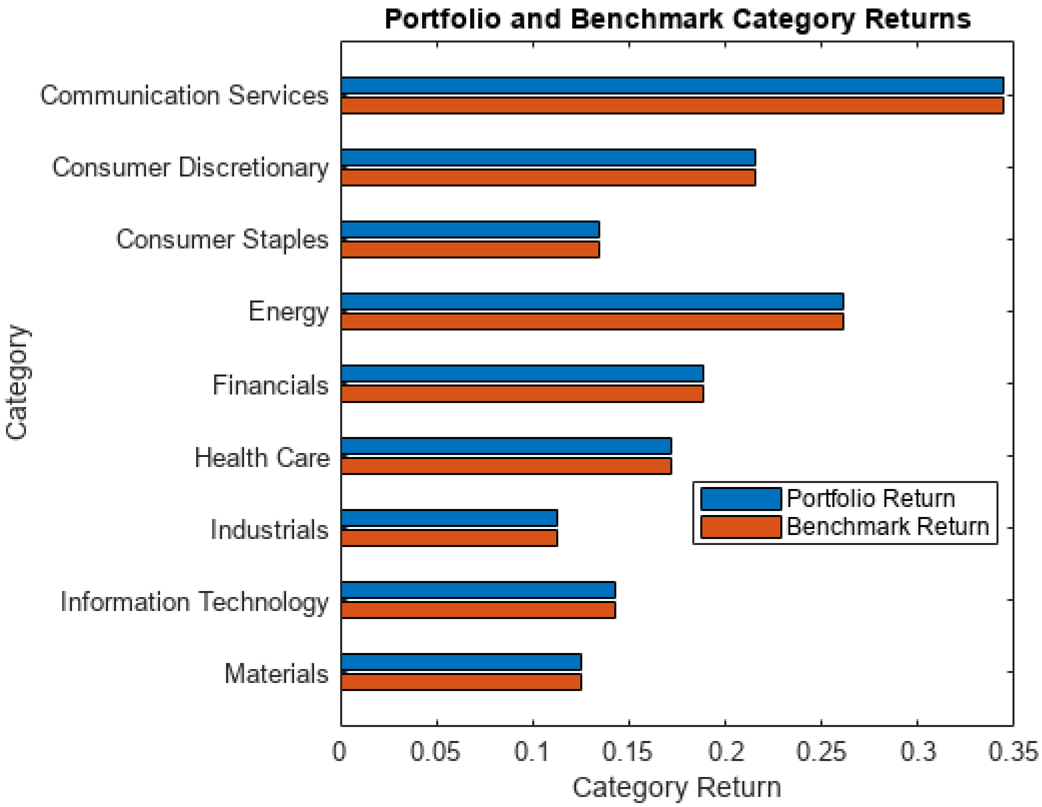
Generate the attribution chart using `attributionsChart` with the `brinson` object. The attributions chart creates a horizontal bar chart of portfolio performance attributions by category, aggregated over all time periods.

```
attributionsChart(brinson)
```



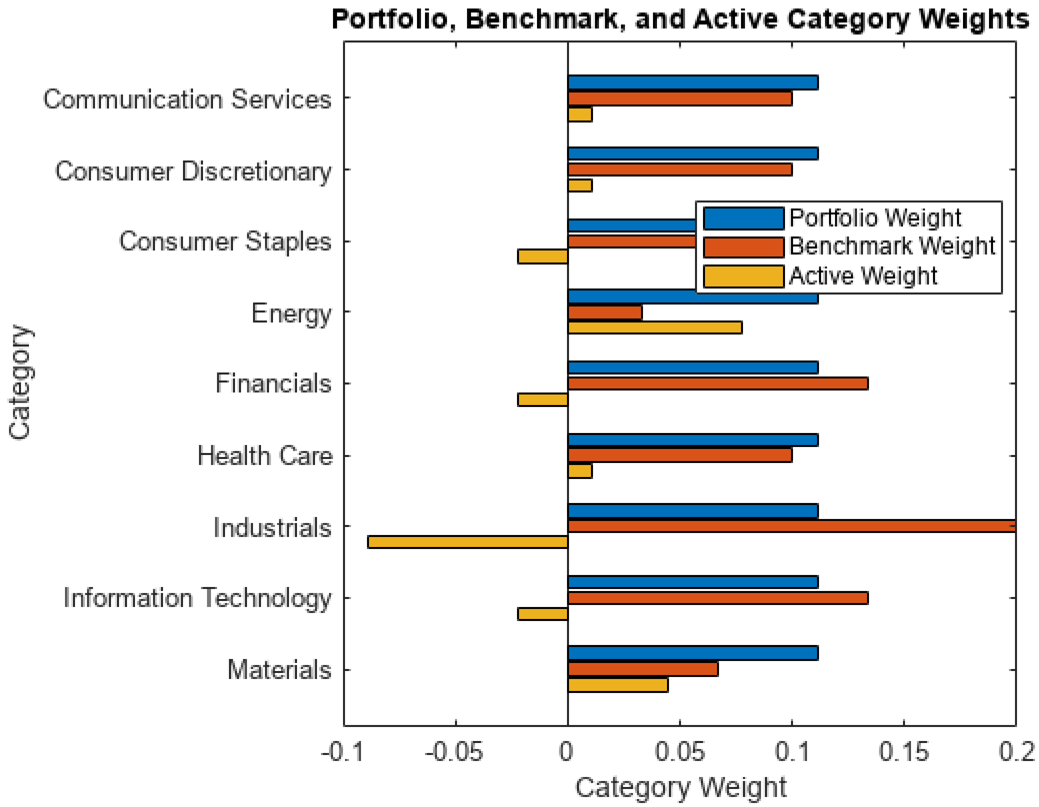
Generate a category returns chart using `categoryReturnsChart` with the `brinson` object. The category returns chart creates a horizontal bar chart of portfolio and benchmark category returns, aggregated over all time periods.

```
categoryReturnsChart(brinson)
```



Generate a category weights chart using `categoryWeightsChart` with the `brinson` object. Te category weights chart creates a horizontal bar chart of portfolio, benchmark, and active weights by category, averaged over all time periods.

```
categoryWeightsChart(brinson)
```



The equal category strategy, despite having different category weights from the benchmark, has the same equal-weighted asset selection process *within each category*. Therefore, the equal category strategy has no selection or interaction effect (relative to the benchmark). Meanwhile, the risk budgeting strategy disregards the asset categories when setting portfolio weights, so it has selection and interaction effects.

Local Functions

Both the equal asset weight and equal category weight rebalance functions compute a fixed weight that is assigned at each rebalance date. In order to avoid calculating the weights each time, the weights are calculated once and then the result is saved in the `UserData` struct for the `backtestStrategy` object. For more information on the `UserData` struct, see the “rebalanceFcn” on page 15-0 input argument for `backtestStrategy`.

```
function [new_weights,user_data] = equalWeightRebalanceFcn(~,pricesTT,user_data)
% Equal asset weight rebalance function

if ~isfield(user_data,'FixedAllocation')
    % If this is the first call to the rebalance function, calculate the
    % desired fixed allocation and save it.
    user_data.FixedAllocation = computeEqualWeights(width(pricesTT));
end
new_weights = user_data.FixedAllocation;

end
```

```
function weights = computeEqualWeights(num_assets)
% Equal asset weight portfolio allocation

weights = ones(1,num_assets) / num_assets;

end


function [new_weights,user_data] = equalCategoryRebalanceFcn(~, ~, user_data)
% Equal category weight rebalance function

if ~isfield(user_data,'FixedAllocation')
    % If this is the first call to the rebalance function, calculate the
    % desired fixed allocation and save it.
    user_data.FixedAllocation = computeEqualCategory(user_data.Categories);
end
new_weights = user_data.FixedAllocation;

end


function weights = computeEqualCategory(asset_categories)
% Equal category weight portfolio allocation

weights = zeros(1,numel(asset_categories));
unique_categories = unique(asset_categories);
category_weight = 1 / numel(unique_categories);

for i = 1:numel(unique_categories)
    category_mask = asset_categories == unique_categories(i);
    weights(category_mask) = category_weight / sum(category_mask);
end

end


function new_weights = riskBudgetingRebalanceFcn(~,pricesTT)
% Risk budgeting rebalance function

new_weights = computeRiskBudgeting(pricesTT);

end


function new_weights = computeRiskBudgeting(pricesTT)
% Risk budgeting portfolio allocation

asset_returns = tick2ret(pricesTT);
asset_cov = cov(asset_returns{:,:});
new_weights = riskBudgetingPortfolio(asset_cov);

end


function asset_table = backtest2brinson(bt,strategy_name,benchmark_name,pricesTT,asset_categories
% Build Brinson attribution input asset table based on the results of the
% completed backtest.
% Compute asset returns per period
asset_returns = tick2ret(pricesTT(period_dates,:));
num_periods = height(asset_returns);

% Brinson Return input
Return = asset_returns.Variables';

% Brinson Category input
Category = repmat(asset_categories,1,num_periods);

% Brinson Period input
num_assets = width(pricesTT);
Period = (1:num_periods) .* ones(num_assets,1);

% Brinson Name input
Name = repmat(string(pricesTT.Properties.VariableNames(:)),1,num_periods);

% Benchmark weights
benchmark_positions = bt.Positions.(benchmark_name){period_dates(1:end-1),2:end}';
BenchmarkWeight = benchmark_positions ./ sum(benchmark_positions);

% Strategy weights
portfolio_positions = bt.Positions.(strategy_name){period_dates(1:end-1),2:end}';
PortfolioWeight = portfolio_positions ./ sum(portfolio_positions);

% Aggregate the inputs into the Brinson asset table
asset_table = table(Period(:),Name(:),Return(:),Category(:), ...
    PortfolioWeight(:),BenchmarkWeight(:),...
    VariableNames=["Period","Name","Return","Category", ...
    "PortfolioWeight","BenchmarkWeight"]);
end
```

See Also

brinsonAttribution | categoryAttribution | categoryReturns | categoryWeights | totalAttribution | summary

Related Examples

- “Backtest Investment Strategies Using Financial Toolbox™” on page 4-231
- “Backtest Investment Strategies with Trading Signals” on page 4-244
- “Backtest Using Risk-Based Equity Indexation” on page 4-285

Analyze Performance Attribution Using Brinson Model

This example shows how to prepare data, create a brinsonAttribution object, and then analyze the performance attribution with respect to category (sector) weights and returns. In this example, you use the categoryAttribution, categoryReturns, categoryWeights, totalAttribution, and summary functions for the analysis. Also, you can generate plots for the results, using categoryReturnsChart, categoryWeightsChart, and attributionsChart.

Prepare Data

Load the stock price data into a table.

```
T = readtable('dowPortfolio.xlsx');
MonthIdx = [1;20;39;62;81;103;125;145;168;188;210;231;251];
MonthlyPrices = T(MonthIdx,3:end);
```

Compute the monthly returns using tick2ret.

```
MonthlyReturns = tick2ret(MonthlyPrices.Variables)';
[NumAssets,NumPeriods] = size(MonthlyReturns);
```

Define the asset categories (sectors) using categorical.

```
Category = categorical(["Materials";"Financials";"Financials"; ...
    "Industrials";"Financials";"Industrials";"Materials"; ...
    "Communication Services";"Industrials";"Consumer Discretionary"; ...
    "Consumer Discretionary";"Industrials";"Information Technology"; ...
    "Information Technology";"Information Technology";"Health Care"; ...
    "Financials";"Consumer Staples";"Consumer Discretionary"; ...
    "Industrials";"Consumer Staples";"Health Care"; ...
    "Information Technology";"Health Care";"Consumer Staples"; ...
    "Communication Services";"Industrials";"Communication Services"; ...
    "Consumer Staples";"Energy"]);
Category = repmat(Category,1,NumPeriods);
```

Define Benchmark and Portfolio Weights

Define the benchmark weight and the portfolio weights, by quarter, for each of the stock assets.

```
BenchmarkWeight = 1./length(MonthlyReturns).*ones(NumAssets, NumPeriods);

PortfolioWeightQ1 = [0;0;0.022;0.033;0;0.044;0.022;0.011;0.065;0.033; ...
    0.055;0.072;0;0.04;0;0.05;0.08;0.042;0.03;0.043;0.055;0.036;0.111; ...
    0.036;0;0.03;0;0.05;0;0.04]*ones(1,3);

PortfolioWeightQ2 = [0;0;0.022;0.033;0;0.044;0.022;0.011;0.049;0.033; ...
    0.055;0.074;0;0.04;0;0.05;0.08;0.042;0.03;0.02;0.055;0.036;0.148; ...
    0.036;0;0.03;0;0.05;0;0.04]*ones(1,3);

PortfolioWeightQ3 = [0;0;0.022;0.033;0;0.042;0.022;0.01;0.049;0.033; ...
    0.05;0.07;0;0.04;0;0.05;0.08;0.042;0.03;0.02;0.055;0.036;0.16; ...
    0.036;0;0.03;0;0.05;0;0.04]*ones(1,3);

PortfolioWeightQ4 = [0;0;0.022;0.033;0;0.042;0.02;0.01;0.039;0.033; ...
    0.05;0.07;0;0.04;0;0.05;0.08;0.042;0.03;0.02;0.055;0.036;0.198; ...
    0.036;0;0.03;0;0.03;0;0.034]*ones(1,3);
```

```
PortfolioWeight = [PortfolioWeightQ1 PortfolioWeightQ2 ...
    PortfolioWeightQ3 PortfolioWeightQ4];

Period = (1:NumPeriods).*ones(NumAssets,1);

Name = repmat(string(MonthlyPrices.Properties.VariableNames(:)),1,NumPeriods);
```

Create AssetTable Input

Use `table` to create the `AssetTable` input to use when creating a `brinsonAttribution` object.

```
AssetTable = table(Period(:), Name(:), ...
MonthlyReturns(:), Category(:), ...
PortfolioWeight(:), BenchmarkWeight(:), ...
VariableNames=["Period", "Name","Return","Category",...
"PortfolioWeight","BenchmarkWeight"])
```

```
AssetTable=360×6 table
    Period     Name        Return              Category            PortfolioWeight    BenchmarkWei

    ______    ______    ___________    ______________________    _______________    _____________

      1       "AA"         0.053621    Materials                            0           0.033333
      1       "AIG"        -0.05964    Financials                           0           0.033333
      1       "AXP"     -0.00019406    Financials                       0.022           0.033333
      1       "BA"        -0.030162    Industrials                      0.033           0.033333
      1       "C"         -0.055015    Financials                           0           0.033333
      1       "CAT"         0.17956    Industrials                      0.044           0.033333
      1       "DD"        -0.090708    Materials                        0.022           0.033333
      1       "DIS"        0.037221    Communication Services           0.011           0.033333
      1       "GE"         -0.07381    Industrials                      0.065           0.033333
      1       "GM"          0.27273    Consumer Discretionary           0.033           0.033333
      1       "HD"        -0.016838    Consumer Discretionary           0.055           0.033333
      1       "HON"        0.025457    Industrials                      0.072           0.033333
      1       "HPQ"        0.083598    Information Technology               0           0.033333
      1       "IBM"       -0.009235    Information Technology            0.04           0.033333
      1       "INTC"        -0.1685    Information Technology               0           0.033333
      1       "JNJ"       -0.066351    Health Care                       0.05           0.033333
      ⋮
```

Create brinsonAttribution Object

Use `brinsonAttribution` to create a `brinsonAttribution` object.

```
BrinsonPAobj = brinsonAttribution(AssetTable)
```

```
BrinsonPAobj = 
  brinsonAttribution with properties:

                NumAssets: 30
       NumPortfolioAssets: 22
       NumBenchmarkAssets: 30
               NumPeriods: 12
            NumCategories: 9
                AssetName: [30x1 string]
              AssetReturn: [30x12 double]
            AssetCategory: [30x12 categorical]
     PortfolioAssetWeight: [30x12 double]
```

```
      BenchmarkAssetWeight: [30x12 double]
   PortfolioCategoryReturn: [9x12 double]
   BenchmarkCategoryReturn: [9x12 double]
   PortfolioCategoryWeight: [9x12 double]
   BenchmarkCategoryWeight: [9x12 double]
           PortfolioReturn: 0.2234
           BenchmarkReturn: 0.2046
              ActiveReturn: 0.0188
```

Compute Category Weights

Use the `brinsonAttribution` object and `categoryWeights` to compute the average and periodic category weights for the portfolio and the benchmark, as well as, the corresponding active weights.

```
[AverageCategoryWeights,PeriodicCategoryWeights] = categoryWeights(BrinsonPAobj)
```

```
AverageCategoryWeights=9×4 table
           Category           AveragePortfolioWeight    AverageBenchmarkWeight    AverageActiveWe
    ______________________    ______________________    ______________________    _______________

    Communication Services            0.0855                      0.1                   -0.0145
    Consumer Discretionary            0.1155                      0.1                    0.0155
    Consumer Staples                   0.097                  0.13333                 -0.036333
    Energy                            0.0385                 0.033333                 0.0051667
    Financials                         0.102                  0.13333                 -0.031333
    Health Care                        0.122                      0.1                     0.022
    Industrials                      0.22375                      0.2                   0.02375
    Information Technology           0.19425                  0.13333                  0.060917
    Materials                         0.0215                 0.066667                 -0.045167

PeriodicCategoryWeights=108×5 table
    Period           Category           PortfolioWeight    BenchmarkWeight    ActiveWeight
    ______    ______________________    _______________    _______________    ____________

      1       Communication Services         0.091                 0.1            -0.009
      1       Consumer Discretionary         0.118                 0.1             0.018
      1       Consumer Staples               0.097             0.13333         -0.036333
      1       Energy                          0.04            0.033333         0.0066667
      1       Financials                     0.102             0.13333         -0.031333
      1       Health Care                    0.122                 0.1             0.022
      1       Industrials                    0.257                 0.2             0.057
      1       Information Technology         0.151             0.13333          0.017667
      1       Materials                      0.022            0.066667         -0.044667
      2       Communication Services         0.091                 0.1            -0.009
      2       Consumer Discretionary         0.118                 0.1             0.018
      2       Consumer Staples               0.097             0.13333         -0.036333
      2       Energy                          0.04            0.033333         0.0066667
      2       Financials                     0.102             0.13333         -0.031333
      2       Health Care                    0.122                 0.1             0.022
      2       Industrials                    0.257                 0.2             0.057
      ⋮
```

Compute Category Returns

Use the `brinsonAttribution` object and `categoryReturns` to compute the aggregate and periodic category (sector) returns for the portfolio and the benchmark.

```
[AggregateCategoryReturns,PeriodicCategoryReturns] = categoryReturns(BrinsonPAobj)
```

```
AggregateCategoryReturns=9×3 table
           Category           AggregatePortfolioReturn    AggregateBenchmarkReturn
    ______________________    ________________________    ________________________

    Communication Services            0.41756                     0.42797         
    Consumer Discretionary            0.27772                     0.34718         
    Consumer Staples                  0.20572                     0.14112         
    Energy                            0.33598                     0.33598         
    Financials                        0.22678                     0.15838         
    Health Care                       0.19451                     0.20679         
    Industrials                       0.14132                     0.14253         
    Information Technology            0.14339                     0.13485         
    Materials                         0.17109                     0.10402         


PeriodicCategoryReturns=108×4 table
    Period           Category           PortfolioReturn    BenchmarkReturn
    ______    ______________________    _______________    _______________

      1       Communication Services        0.056262           0.052385   
      1       Consumer Discretionary        0.079767            0.10017   
      1       Consumer Staples              -0.01485         -0.0033588   
      1       Energy                        0.073093           0.073093   
      1       Financials                  -0.0021179          -0.029374   
      1       Health Care                   0.012124           0.022297   
      1       Industrials                  0.0018686          0.0088589   
      1       Information Technology        0.033481          -0.011315   
      1       Materials                    -0.090708          -0.018543   
      2       Communication Services        0.069253           0.078033   
      2       Consumer Discretionary       -0.023091          -0.036416   
      2       Consumer Staples             0.0027749         0.00096619   
      2       Energy                         -0.0487            -0.0487   
      2       Financials                    0.033379            0.02054   
      2       Health Care                   0.014975           0.015929   
      2       Industrials                   0.047732           0.041906   
      ⋮
```

Compute Category Attribution

Use the `brinsonAttribution` object and `categoryAttribution` to compute the performance attribution of the portfolio for each category (sector).

```
[AggregateCategoryAttribution,PeriodicCategoryAttribution] = categoryAttribution(BrinsonPAobj)
```

```
AggregateCategoryAttribution=9×5 table
           Category           Allocation     Selection     Interaction    ActiveReturn
    ______________________    __________    ___________    ___________    ____________

    Communication Services    -0.0025318    -0.00071062      0.0002061     -0.0030363 
    Consumer Discretionary     0.0024505       -0.00678     -0.0011609     -0.0054903 
    Consumer Staples           0.0023692      0.0093618     -0.0025511        0.00918 
    Energy                    0.00038824     1.3872e-19     2.7745e-20     0.00038824 
    Financials                 0.0013464      0.0097268     -0.0022858      0.0087874 
    Health Care               0.00011567     -0.0013031    -0.00028667     -0.0014741 
    Industrials                0.0016701    -0.00035051    -5.7623e-05       0.001262 
    Information Technology     0.0015686      0.0017812     0.00016898      0.0035187 
    Materials                  0.0040725      0.0050687     -0.0034678      0.0056734

PeriodicCategoryAttribution=108×7 table
    Period    LinkingCoefficient           Category           Allocation      Selection     Inter
    ______    __________________    ______________________    ___________    ___________    _____

      1             1.1901          Communication Services    -0.00034021     0.00038766     -3.4
      1             1.1901          Consumer Discretionary      0.0015405     -0.0020403    -0.00
      1             1.1901          Consumer Staples           0.00065192     -0.0015322     0.00
      1             1.1901          Energy                     0.00039006              0
      1             1.1901          Financials                  0.0013774      0.0036342    -0.00
      1             1.1901          Health Care                0.00016968     -0.0010173     -0.0
      1             1.1901          Industrials               -0.00032634     -0.0013981    -0.00
      1             1.1901          Information Technology    -0.00045756      0.0059728      0.0
      1             1.1901          Materials                   0.0014797      -0.004811      0.0
      2             1.1935          Communication Services    -0.00058016      -0.000878      7.9
      2             1.1935          Consumer Discretionary    -0.00089975      0.0013325     0.00
      2             1.1935          Consumer Staples           0.00045794     0.00024116    -6.57
      2             1.1935          Energy                    -0.00041514              0
      2             1.1935          Financials                -0.00021838      0.0017119     -0.0
      2             1.1935          Health Care                  5.19e-05    -9.5418e-05    -2.09
      2             1.1935          Industrials                 0.0016152      0.0011651     0.00
      ⋮
```

Compute Total Attribution

Use the `brinsonAttribution` object and `totalAttribution` to compute the total performance attribution of the portfolio summed up for all categories (sectors).

```
[AggregateTotalAttribution,PeriodicTotalAttribution] = totalAttribution(BrinsonPAobj)
```

```
AggregateTotalAttribution=1×4 table
    Allocation    Selection    Interaction    ActiveReturn
    __________    _________    ___________    ____________

     0.011449     0.016794     -0.0094348       0.018809

PeriodicTotalAttribution=12×6 table
    Period    LinkingCoefficient    Allocation      Selection     Interaction    ActiveReturn
    ______    __________________    ___________    ___________    ___________    ____________

       1            1.1901            0.0044852    -0.00080413      0.0025538      0.0062349
       2            1.1935           0.00087285      0.0032617     -0.0025973      0.0015372
       3            1.1904          -6.9025e-05      0.0052394     0.00069386      0.0058643
       4             1.208            -0.004707     -0.0095748     -0.0046622      -0.018944
       5            1.1995          -0.00030704      -0.006091    -0.00061525     -0.0070132
       6            1.1935            0.0012025    -0.00026724     0.00058932      0.0015246
       7            1.1813            0.0032236       0.017541     -0.0021062       0.018658
       8             1.196             0.004629      -0.004929     -0.0018038     -0.0021038
       9            1.1927            0.0013046     0.00036681     0.00099195      0.0026634
      10            1.1923            0.0032172      0.0023077     -0.0023663      0.0031586
      11             1.194            -0.001881     0.00044958      0.0022018     0.00077039
      12            1.1918           -0.0022819      0.0069324     -0.0007375       0.003913
```

Generate Summary for Performance Attribution

Use the `brinsonAttribution` object and `summary` to generate a table that summarizes the final results (aggregated over all time periods and categories) of the performance attribution using the Brinson model.

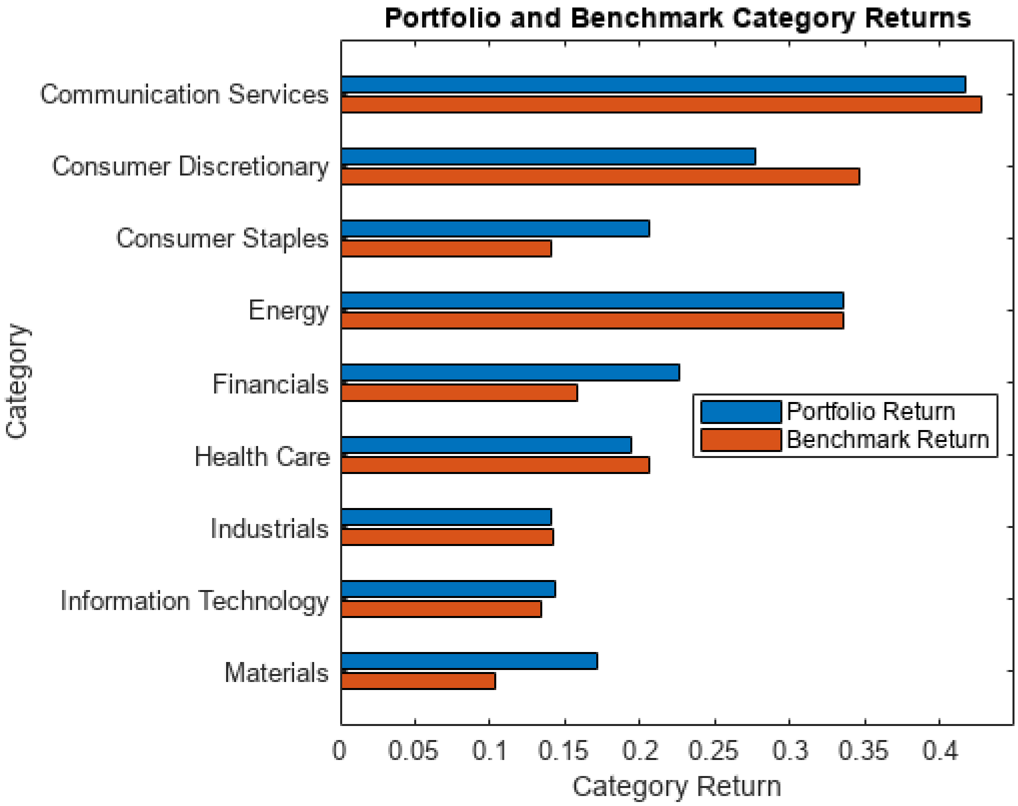
```
SummaryTable = summary(BrinsonPAobj)
```

```
SummaryTable=11×1 table
                                     Brinson Attribution Summary
                                     ___________________________

    Total Number of Assets                         30
    Number of Assets in Portfolio                  22
    Number of Assets in Benchmark                  30
    Number of Periods                              12
    Number of Categories                            9
    Portfolio Return                          0.22345
    Benchmark Return                          0.20464
    Active Return                            0.018809
    Allocation Effect                        0.011449
    Selection Effect                         0.016794
    Interaction Effect                     -0.0094348
```

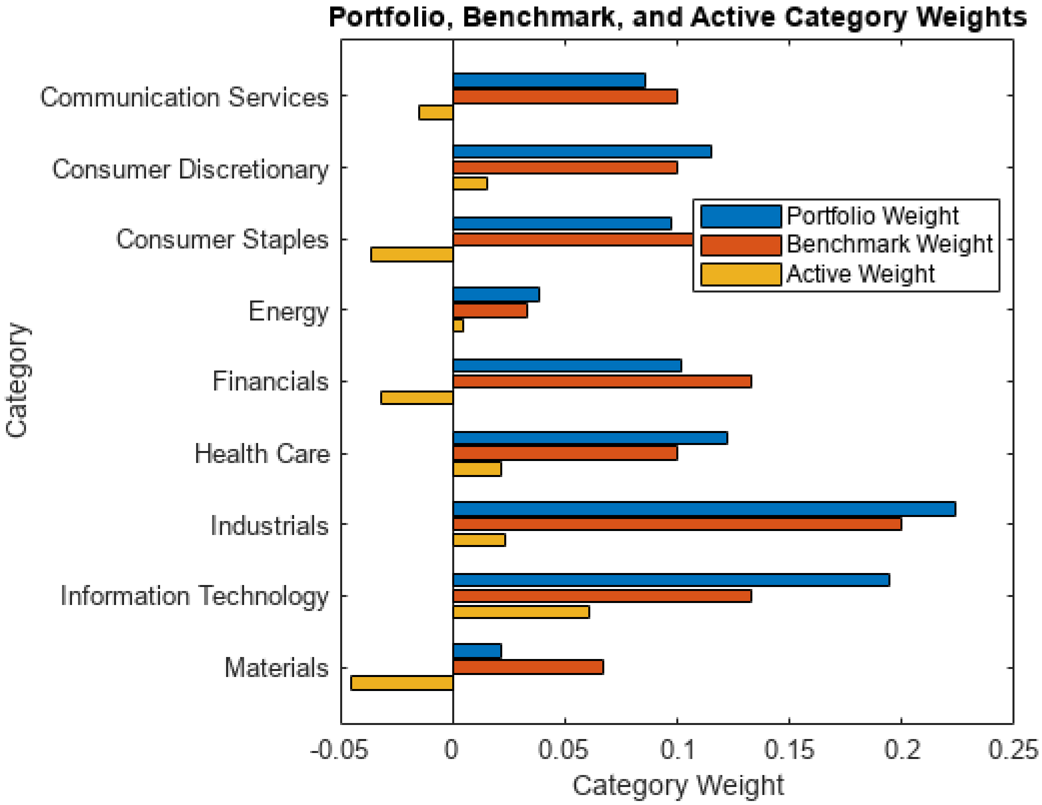
Generate Horizontal Bar Charts for Returns, Weights, and Performance Attribution

Use the `brinsonAttribution` object and `categoryReturnsChart`, `categoryWeightsChart`, and `attributionsChart` to generate horizontal bar charts.

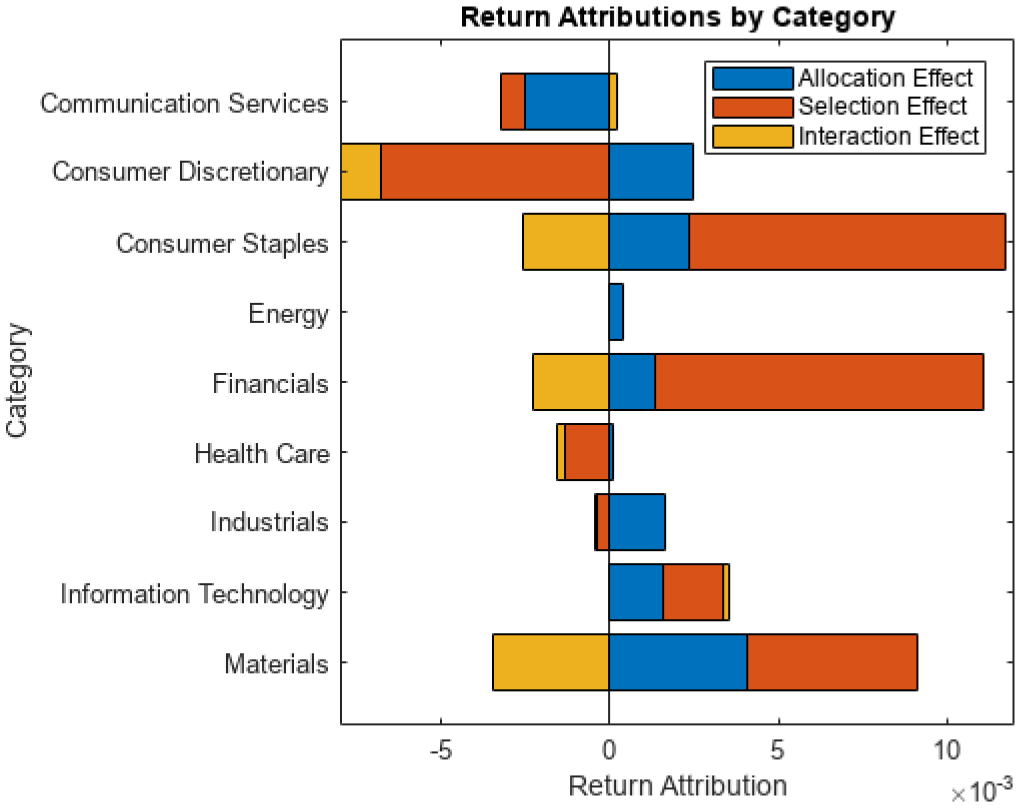
```
categoryReturnsChart(BrinsonPAobj)
```



```
categoryWeightsChart(BrinsonPAobj)
```



```
attributionsChart(BrinsonPAobj, Style="stacked")
```



See Also

`brinsonAttribution` | `categoryAttribution` | `categoryReturns` | `categoryWeights` | `totalAttribution` | `summary`

Related Examples

- "Backtest with Brinson Attribution to Evaluate Portfolio Performance" on page 4-311

Diversify Portfolios Using Custom Objective

This example shows three techniques of asset diversification in a portfolio using the `estimateCustomObjectivePortfolio` function with a `Portfolio` object. The purpose of asset diversification is to balance the exposure of the portfolio to any given asset to reduce volatility over a period of time. Given the sensitivity of the minimum variance portfolio to the estimation of the covariance matrix, some practitioners have added diversification techniques to the portfolio selection with the hope of minimizing risk measures other than the variance measures such as turnover, maximum drawdown, and so on.

This example applies these common diversification techniques:

- Equally weighted (EW) portfolio on page 4-331
- Equal risk contribution (ERC) on page 4-334
- Most diversified portfolio (MDP) on page 4-332

Additionally, this example demonstrates penalty methods that you can use to achieve different degrees of diversification. In those methods, you add a penalty term to the `estimateCustomObjectivePortfolio` function to balance the level of variance reduction and the diversification of the portfolio.

Retrieve Market Data and Define Mean-Variance Portfolio

Begin by loading and computing the expected returns and their covariance matrix.

```
% Load data
load('port5.mat');
% Store returns and covariance
mu = mean_return;
Sigma = Correlation .* (stdDev_return * stdDev_return');
```

Define a mean-variance portfolio using a `Portfolio` object with default constraints to create a fully invested, long-only portfolio.

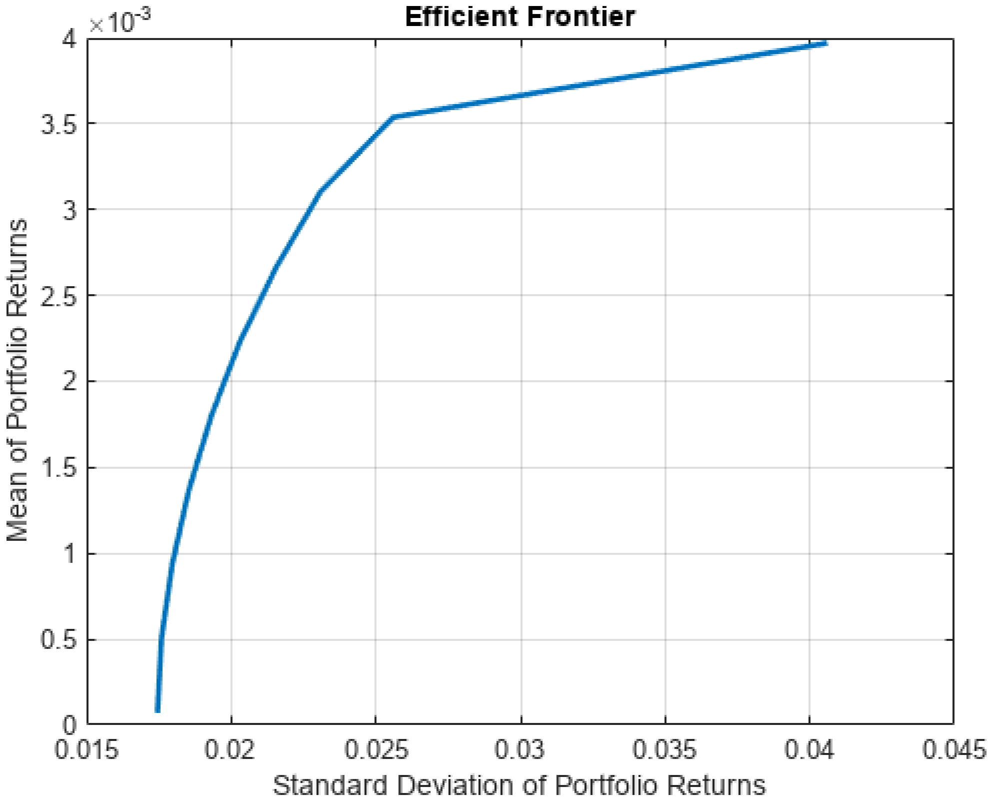
```
% Create a mean-variance Portfolio object with default constraints
p = Portfolio('AssetMean',mu,'AssetCovar',Sigma);
p = setDefaultConstraints(p);
```

One of the many features of the `Portfolio` object is that it can obtain the efficient frontier of the portfolio problem. The efficient frontier is computed by solving a series of optimization problems for which the return level of the portfolio, μ_0, is modified to obtain different points on the efficient frontier. These problems are defined as

$$\begin{aligned} \min \;\; & x^T \Sigma x \\ \text{st.} \;\; & \sum_{i=1}^{n} x_i = 1 \\ & x^T \mu \geq \mu_0 \\ & x \geq 0 \end{aligned}$$

The advantage of using the `Portfolio` object to compute the efficient frontier is that you can obtain without having to manually formulate and solve the multiple optimization problems shown above.

```
plotFrontier(p);
```



The Portfolio object can also compute the weights associated with the minimum variance portfolio, which is defined by the following problem.

$$\begin{aligned} &\min \ x^T \Sigma x \\ &\text{st.} \ \sum_{i=1}^{n} x_i = 1 \\ &\quad x \geq 0 \end{aligned}$$

The minimum variance weights is the benchmark against which all the weights of the diversification strategies are compared.

```
wMinVar = estimateFrontierLimits(p,'min');
```

To learn more about the problems that you can solve with the Portfolio object, see “When to Use Portfolio Objects Over Optimization Toolbox” on page 4-128.

Specify Diversification Techniques

This section presents the three diversification methods. Each of the three diversification methods is associated with a diversification measure and that diversification measure defines a penalty term to achieve different diversification levels. The diversification, obtained by adding the penalty term to the objective function, ranges from the behavior achieved by the minimum variance portfolio to the behavior of the EW, ECR, and MDP, respectively.

The default portfolio has only one equality constraint and a lower bound for the assets weights. The weights must be nonnegative and they must sum to 1. The feasible set is represented as$X$:

$$X = \left\{ x \;\middle|\; x \geq 0,\ \sum_{i=1}^{n} x_i = 1 \right\}$$

Equally Weighted Portfolio

One of the diversification measures is the Herfindahl-Hirschman (HH) index defined as:

$$\mathrm{HH}(x) = \sum_{i=1}^{n} x_i^2$$

This index is minimized when the portfolio is equally weighted. The portfolios obtained from using this index as a penalty have weights that satisfy the portfolio constraints and that are more evenly weighted.

The portfolio that minimizes the HH index is $\min_{x \in X}\ x^T x$. Since the constraints in X are the default constraints, the solution of this problem is the equally weighted (EW) portfolio. If X had extra constraints, the solution would be the portfolio that satisfies all the constraints and, at the same time, keeps the weights as equal as possible. Use the function handle HHObjFun for the Herfindahl-Hirschman (HH) index with the estimateCustomObjectivePortfolio function.

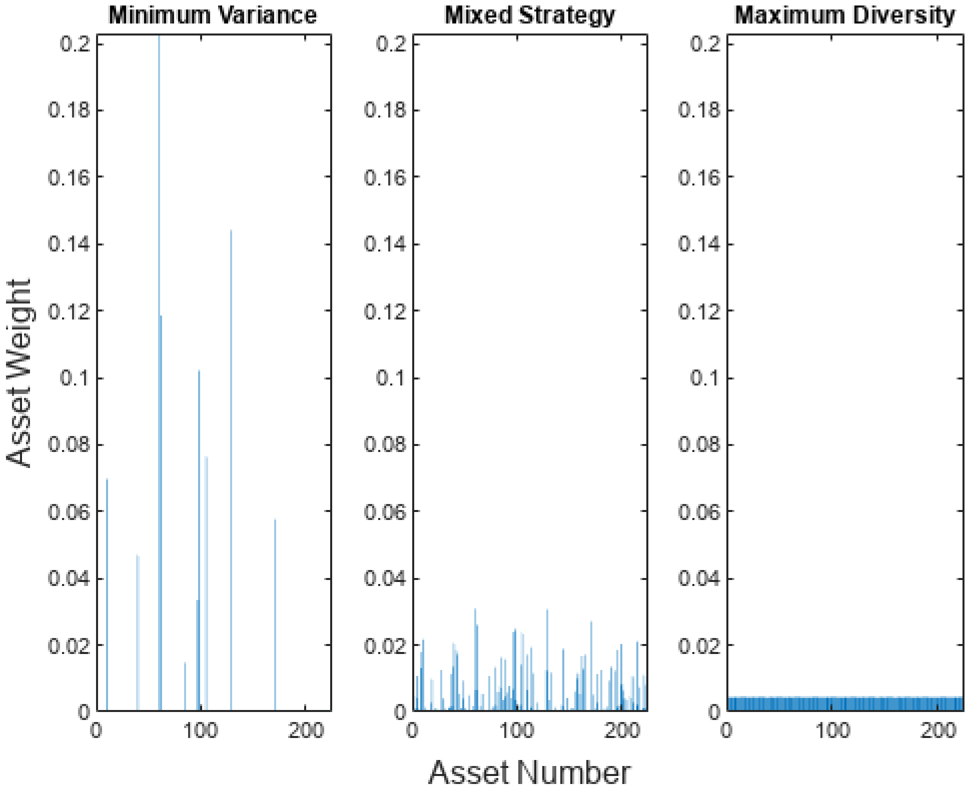
```
% Maximize the HH diversification (by minimizing the HH index)
HHObjFun = @(x) x'*x;
% Solution that minimizes the HH index
wHH = estimateCustomObjectivePortfolio(p,HHObjFun);
```

The portfolio that minimizes the variance with the HH penalty is $\min_{x \in X}\ x^T \Sigma x + \lambda_{\mathrm{HH}} x^T x$. Use the function handle HHMixObjFun for the HH penalty with the estimateCustomObjectivePortfolio function.

```
% HH penalty parameter
lambdaHH = 1e-2;
% Variance + Herfindahl-Hirschman (HH) index
HHMixObjFun = @(x) x'*p.AssetCovar*x + lambdaHH*(x'*x);
% Solution that accounts for risk and HH diversification
wHHMix = estimateCustomObjectivePortfolio(p,HHMixObjFun);
```

Plot the weights distribution for the minimum variance portfolio, the equal weight portfolio, and the penalized strategy.

```
% Plot different strategies
plotAssetAllocationChanges(wMinVar,wHHMix,wHH)
```



This plot shows how the penalized strategy returns weights that are between the minimum variance portfolio and the EW portfolio. In fact, choosing $\lambda_{\mathrm{HH}} = 0$ returns the minimum variance solution, and as $\lambda_{\mathrm{HH}} \rightarrow \infty$, the solution approaches the EW portfolio.

Most Diversified Portfolio

The diversification index associated to the most diversified portfolio (MDP) is defined as

$$\mathrm{MDP}(x) = -\sum_{i=1}^{n} \sigma_i x_i$$

where σ_i represents the standard deviation of asset i.

The MDP is the portfolio that maximizes the diversification ratio:

$$\varphi(x) = \frac{x^T \sigma}{\sqrt{x^T \Sigma x}}$$

The diversification ratio $\varphi(x)$ is equal to 1 if the portfolio is fully invested in one asset or if all assets are perfectly correlated. For all other cases, $\varphi(x) > 1$. If $\varphi(x) \approx 1$, there is no diversification, so the goal is to find the portolio that maximizes $\varphi(x)$:

$$\max_{x \in X} \frac{\sigma^T x}{\sqrt{x^T \Sigma x}}$$

Unlike the HH index, the MDP goal is not to obtain a portfolio whose weights are evenly distributed among all assets, but to obtain a portfolio whose selected (nonzero) assets have the same correlation to the portfolio as a whole. Use the function handle `MDPObjFun` for the most diversified portfolio (MDP) with the `estimateCustomObjectivePortfolio` function.

```
% Maximize the diversification ratio
sigma = sqrt(diag(p.AssetCovar));
MDPObjFun = @(x) (sigma'*x)/sqrt(x'*p.AssetCovar*x);
% Solution of MDP
wMDP = estimateCustomObjectivePortfolio(p,MDPObjFun, ...
    ObjectiveSense="maximize");
```

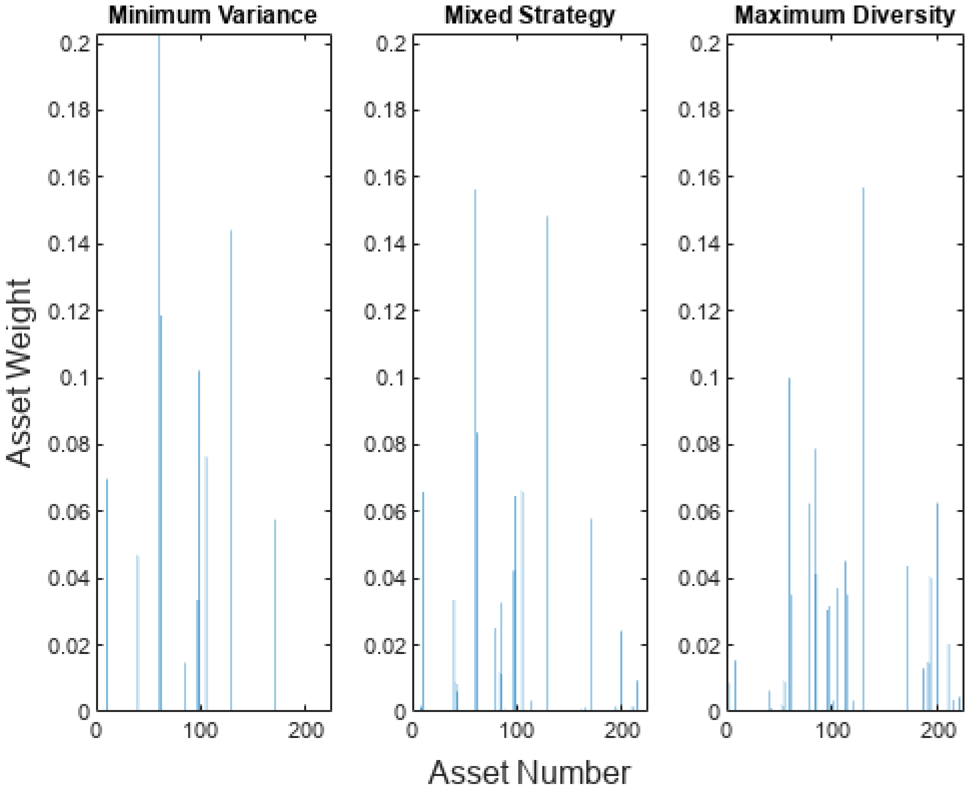
The following code shows that there exists a value $\widehat{\lambda}_{\text{MDP}} > 0$ such that the MDP problem and the problem with its penalized version are equivalent.The portfolio that minimizes the variance with the MDP penalty is $\min_{x \in X} \ x^T \Sigma x - \lambda_{\text{MDP}} \, \sigma^T x$.

Define an MDP penalty parameter and solve for MDP using the function handle `MDPMixObjFun` for the MDP penalty with the `estimateCustomObjectivePortfolio` function.

```
% MDP penalty parameter
lambdaMDP = 1e-2;
% Variance + Most Diversified Portfolio (MDP)
MDPMixObjFun = @(x) x'*p.AssetCovar*x - lambdaMDP*(sigma'*x);
% Solution that accounts for risk and MDP diversification
wMDPMix = estimateCustomObjectivePortfolio(p,MDPMixObjFun);
```

Plot the weights distribution for the minimum variance portfolio, the MDP, and the penalized strategy.

```
% Plot different strategies
plotAssetAllocationChanges(wMinVar,wMDPMix,wMDP)
```



In this plot, the penalized strategy weights are between the minimum variance portfolio and the MDP. This result is the same as with the HH penalty, where choosing $\lambda_{\text{MDP}} = 0$ returns the minimum variance solution and values of $\lambda_{\text{MDP}} \in \left[0, \widehat{\lambda}_{\text{MDP}}\right]$ return asset weights that range from the minimum variance behavior to the MDP behavior.

Equal Risk Contribution Portfolio

The diversification index associated with the equal risk contribution (ERC) portfolio is defined as

$$\text{ERC}(x) = -\sum_{i=1}^{n} \ln(x_i)$$

This index is related to a convex reformulation shown by Maillard [1 on page 4-339] that computes the ERC portfolio. The authors show that you can obtain the ERC portfolio by solving the following optimization problem

$$\min_{y \geq 0} y^T \Sigma y$$
$$\text{st.} \quad \sum_{i=1}^{n} \ln(y_i) \geq c$$

and by defining x, the ERC portfolio with default constraints, as $x = \frac{y}{\Sigma_i y_i}$, where $c > 0$ can be any constant. You implement this procedure in the riskBudgetingPortfolio on page 4-339 function.

The purpose of the ERC portfolio is to select the assets weights in such a way that the risk contribution of each asset to the portfolio volatility is the same for all assets.

```
% ERC portfolio
wERC = riskBudgetingPortfolio(p.AssetCovar);
```

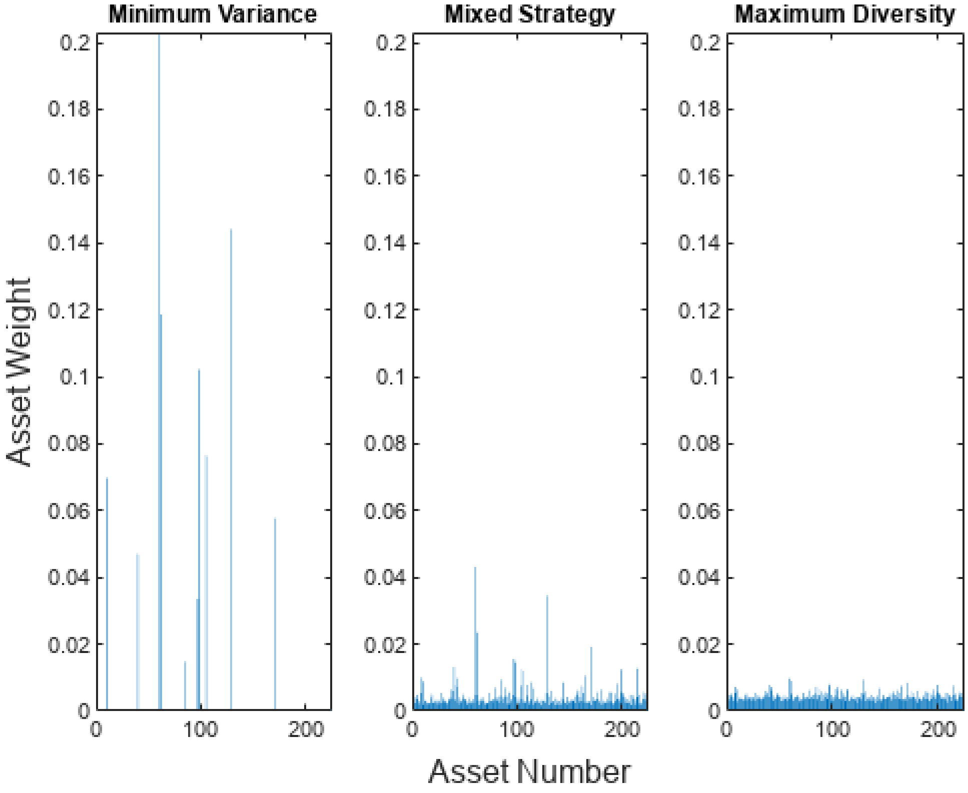
The portfolio that minimizes the variance with the ERC penalty is $\min_{x \in X} \; x^T \Sigma x - \lambda_{\mathrm{ERC}} \sum_{i=1}^{n} \ln(x_i)$.

Similar to the case for the MDP penalized formulation, there exists a $\hat{\lambda}_{\mathrm{ERC}}$ such that the ERC problem and its penalized version are equivalent. Use the function handle `ERCMixObjFun` for the ERC penalty with the `estimateCustomObjectivePortfolio` function.

```
% ERC penalty parameter
lambdaERC = 3e-6; % lambdaERC is so small because the log of a number
                  % close to zero (the portfolio weights) is large
% Variance + Equal Risk Contribution (ERC)
ERCMixObjFun = @(x) x'*p.AssetCovar*x - lambdaERC*sum(log(x));
% Solution that accounts for risk and ERC diversification
wERCMix = estimateCustomObjectivePortfolio(p,ERCMixObjFun);
```

Plot the weights distribution for the minimum variance portfolio, the ERC, and the penalized strategy.

```
% Plot different strategies
plotAssetAllocationChanges(wMinVar,wERCMix,wERC)
```



Comparable to the two diversification measures above, here the penalized strategy weights are between the minimum variance portfolio and the ERC portfolio. Choosing $\lambda_{\text{ERC}} = 0$ returns the minimum variance solution and the values of $\lambda_{\text{ERC}} \in \left[0, \widehat{\lambda}_{\text{ERC}}\right]$ return asset weights that range from the minimum variance behavior to the ERC portfolio behavior.

Compare Diversification Strategies

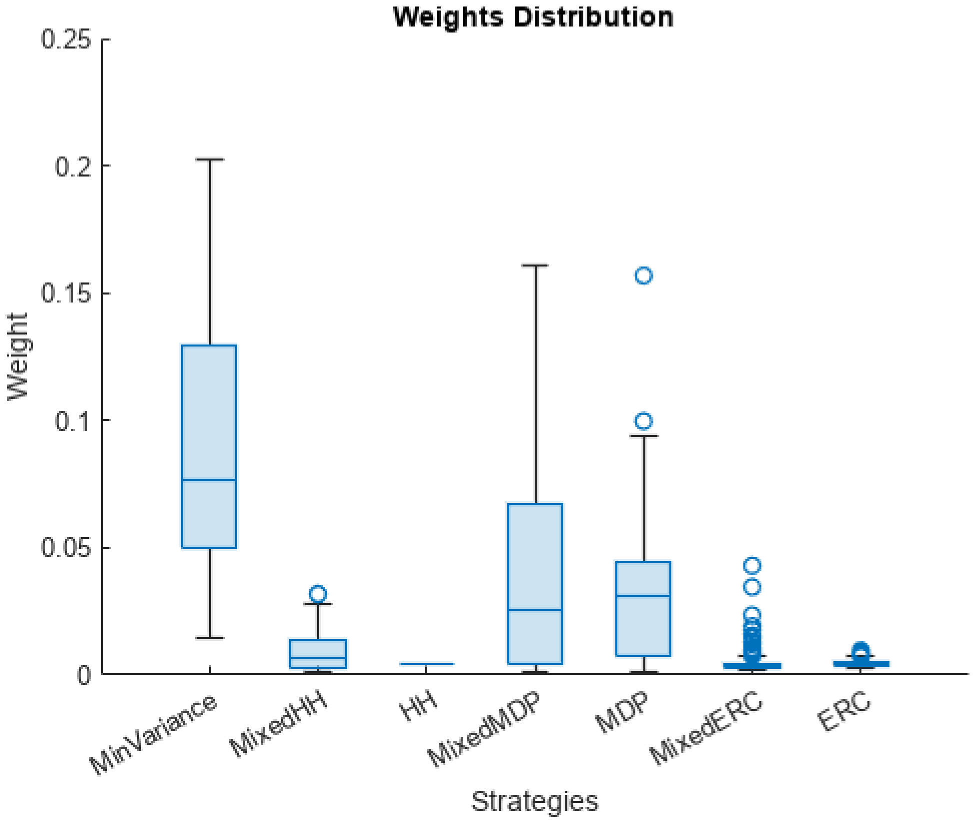
Compute the number of assets that are selected in each portfolio. Assume that an asset is selected if the weight associated to that asset is above a certain threshold.

```
% Build a weights table
varNames = {'MinVariance','MixedHH','HH','MixedMDP','MDP', ...
    'MixedERC','ERC'};
weightsTable = table(wMinVar,wHHMix,wHH,wMDPMix,wMDP, ...
    wERCMix,wERC,'VariableNames',varNames);
% Number of assets with nonzero weights
cutOff = 1e-3; % Weights below cut-off point are considered zero.
[reweightedTable,TnonZero] = tableWithNonZeroWeights(weightsTable, ...
    cutOff,varNames);
display(TnonZero)
```

```
TnonZero=1×7 table
                       MinVariance    MixedHH    HH     MixedMDP    MDP    MixedERC    ERC
                       ___________    _______    ___    ________    ___    ________    ___

    Nonzero weights        11           104      225       23       28       225       225
```

As discussed above, the HH penalty goal is to obtain more evenly weighted portfolios. The portfolio that maximizes the HH diversity (and corresponds to the EW portfolio when only the default constraints are selected) has the largest number of assets selected and the weights of these assets are closer together. You can see this latter quality in the following `boxchart`. Also, the strategy that adds the HH index as a penalty function to the objective has a larger number of assets than the minimum variance portfolio but less than the portfolio that maximizes HH diversity. The ERC portfolio also selects all the assets because all weigths need to be nonzero to have some risk contribution.

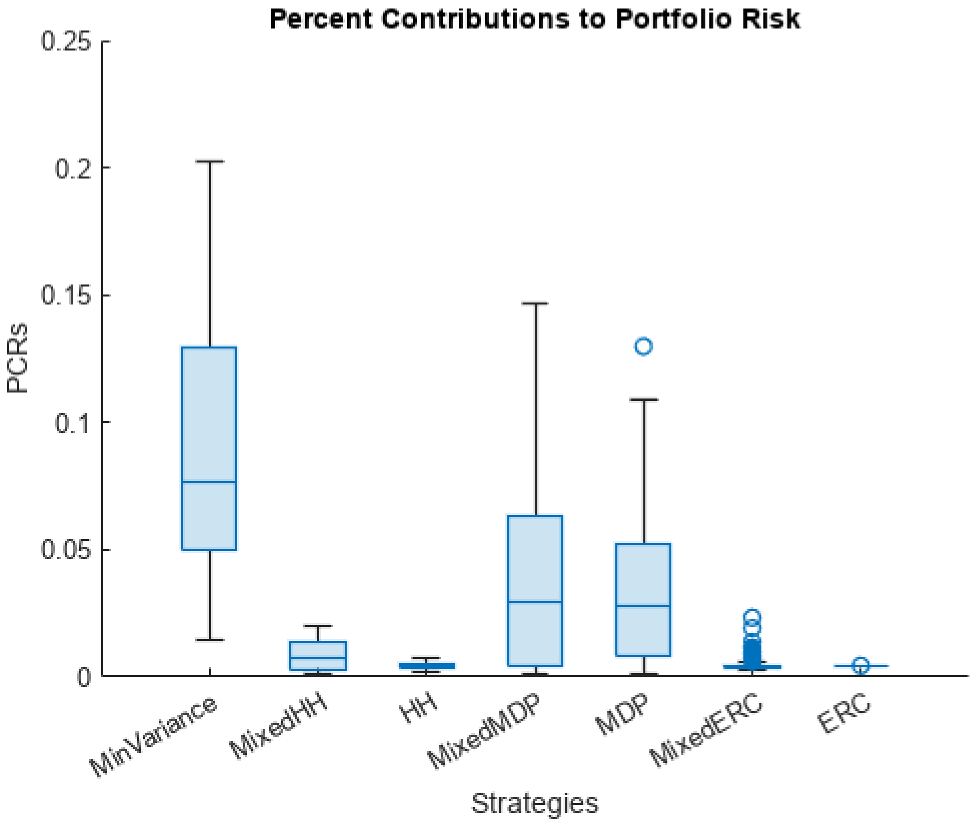
```
% Boxchart of portfolio weights
figure;
matBoxPlot = reweightedTable.Variables;
matBoxPlot(matBoxPlot == 0) = NaN;
boxchart(matBoxPlot)
xticklabels(varNames)
title('Weights Distribution')
xlabel('Strategies')
ylabel('Weight')
```



This `boxchart` shows the spread of the assets' positive weights for the different portfolios. As previously discussed, the weights of the portfolio that maximize the HH diversity are all the same. If the portfolio had other types of constraints, the weights would not all be the same, but they would have the lowest variance. The ERC portfolio weights also have a small variance. In fact, you can observe as the number of assets increases as the variance of the ERC portfolio weights becomes smaller.

The weights variability of the MDP is smaller than the variability of the minimum variance weights. However, it is not necessarily true that the MDP's weights will have less variability than the minimum variance weights because the goal of the MDP is not to obtain equally weighted assets, but to distribute the correlation of each asset with its portfolio evenly.

```
% Compute and plot the risk contribution of each individual
% asset to the portfolio
riskContribution = portfolioRiskContribution(p.AssetCovar, ...
    weightsTable.Variables);
% Remove small contributions
riskContribution(riskContribution < 1e-3) = NaN;

% Compare percent contribution to portofolio risk
boxchart(riskContribution)
xticklabels(varNames)
title('Percent Contributions to Portfolio Risk')
xlabel('Strategies')
ylabel('PCRs')
```

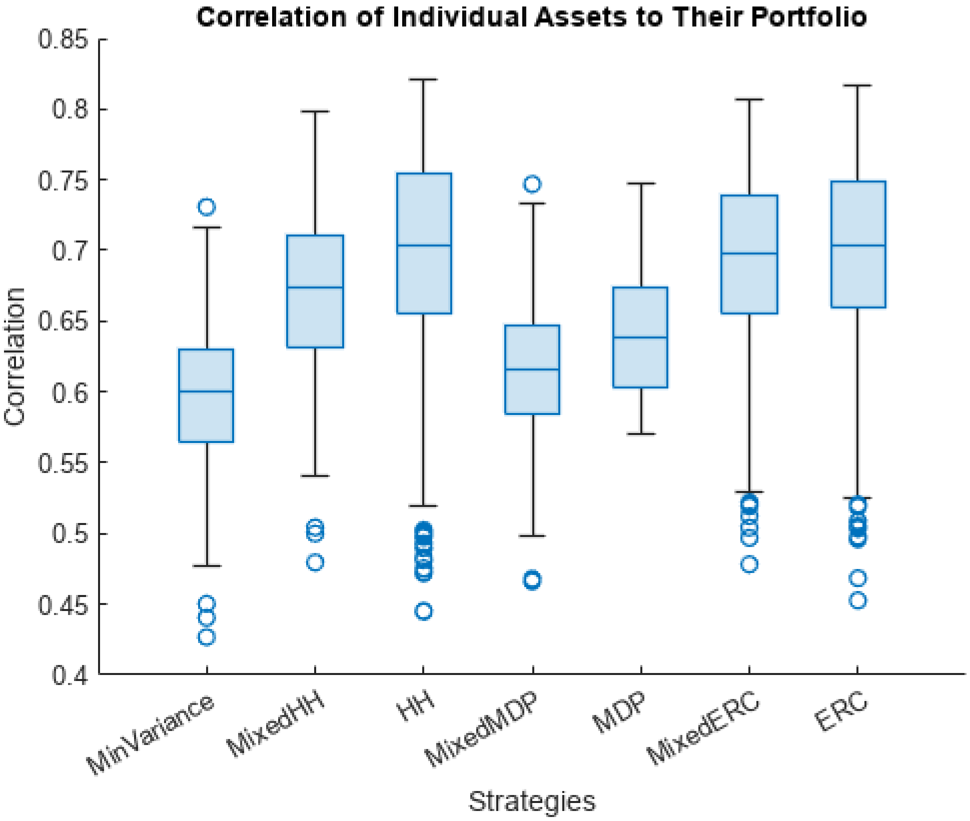


This `boxchart` shows the percent risk contribution of each asset to the total portfolio risk. The percent risk contribution is computed as

$$(\mathrm{PRC})_i = \frac{x_i(\Sigma x)_i}{x^T \Sigma x}$$

As expected, all the ERC portfolio assets have the same risk contribution to the portfolio. As discussed after the weights distribution plot, if the problem had other types of constraints, the risk contribution of the ERC portfolio would not be the same for all assets, but they would have the lowest variance. Also, the behavior shown in this picture is similar to the behavior shown by the weights distribution.

```
% Compute and plot the correlation of each individual asset to its
% portfolio
corrAsset2Port = correlationInfo(p.AssetCovar, ...
    weightsTable.Variables);
% Boxchart of assets to portfolio correlations
figure
boxchart(corrAsset2Port)
xticklabels(varNames)
title('Correlation of Individual Assets to Their Portfolio')
xlabel('Strategies')
ylabel('Correlation')
```



This `boxchart` shows the distribution of the correlations of each asset with its respective portfolio. The correlation of asset *i* to its portfolio is computed with the following formula:

$$\rho_{\mathrm{iP}} = \frac{(\Sigma x)_i}{\sigma_i \sqrt{x^T \Sigma x}}$$

The MDP is the portfolio whose correlations are closer together and this is followed by the strategy that uses the MDP penalty term. In fact, if the portfolio problem allowed negative weights, then all the assets of the MDP would have the same correlation to its portfolio. Also, both the HH (EW) and ERC portfolios have almost the same correlation variability.

References

1 Maillard, S., Roncalli, T., & Teïletche, J. "The Properties of Equally Weighted Risk Contribution Portfolios." *The Journal of Portfolio Management*, *36*(4), 2010, pp. 60–70.

2 Richard, J. C., & Roncalli, T. "Smart Beta: Managing Diversification of Minimum Variance Portfolios." *Risk-Based and Factor Investing*. Elsevier, 2015, pp. 31–63.

3 Tütüncü, R., Peña, J., Cornuéjols, G. *Optimization Methods in Finance*. United Kingdom: Cambridge University Press, 2018.

Local Functions

```
function [] = plotAssetAllocationChanges(wMinVar,wMix,wMaxDiv)
% Plots the weights allocation from the strategies shown before
```

```
figure
t = tiledlayout(1,3);
nexttile
bar(wMinVar')
axis([0 225 0 0.203])
title('Minimum Variance')
nexttile
bar(wMix')
axis([0 225 0 0.203])
title('Mixed Strategy')
nexttile
bar(wMaxDiv')
axis([0 225 0 0.203])
title('Maximum Diversity')
ylabel(t,'Asset Weight')
xlabel(t,'Asset Number')

end

function [weightsTable,TnonZero] = ...
    tableWithNonZeroWeights(weightsTable,cutOff,varNames)
% Creates a table with the number of nonzero weights for each strategy

% Select only meaningful weights
funSelect = @(x) (x >= cutOff).*x./sum(x(x >= cutOff));
weightsTable = varfun(funSelect,weightsTable);

% Number of assets with positive weights
funSum = @(x) sum(x > 0);
TnonZero = varfun(funSum,weightsTable);
TnonZero.Properties.VariableNames = varNames;
TnonZero.Properties.RowNames = {'Nonzero weights'};

end

function [corrAsset2Port] = correlationInfo(Sigma,portWeights)
% Returns a matrix with the correlation of each individual asset to its
% portfolio

nX = size(portWeights,1); % Number of assets
nP = size(portWeights,2); % Number of portfolios

auxM = eye(nX);
corrAsset2Port = zeros(nX,nP);
for j = 1:nP
    % Portfolio's standard deviation
    sigmaPortfolio = sqrt(portWeights(:,j)'*Sigma*portWeights(:,j));
    for i = 1:nX
        % Assets's standard deviation
        sigmaAsset = sqrt(Sigma(i,i));
        % Asset to portfolio correlation
        corrAsset2Port(i,j) = (auxM(:,i)'*Sigma*portWeights(:,j))/...
            (sigmaAsset*sigmaPortfolio);
    end
end

end
```

```
function [riskContribution] = portfolioRiskContribution(Sigma,...
    portWeights)
% Returns a matrix with the risk contribution of each asset to
% the underlying portfolio.

nX = size(portWeights,1); % Number of assets
nP = size(portWeights,2); % Number of portfolios

riskContribution = zeros(nX,nP);
for i = 1:nP
    weights = portWeights(:,i);
    % Portfolio variance
    portVar = weights'*Sigma*weights;
    % Marginal constribution to portfoli risk (MCR)
    margRiskCont = weights.*(Sigma*weights)/sqrt(portVar);
    % Percent contribution to portfolio risk
    riskContribution(:,i) = margRiskCont/sqrt(portVar);
end

end
```

See Also

estimatePortSharpeRatio | estimateFrontier | estimateFrontierByReturn | estimateFrontierByRisk

Related Examples

- “Diversify ESG Portfolios” on page 4-265
- “Diversify Portfolios Using Optimization Toolbox”

More About

- “Portfolio Optimization Theory” on page 4-3

Solve Problem for Minimum Variance Portfolio with Tracking Error Penalty

This example shows how to compute a portfolio that minimizes the tracking error subject to a benchmark portfolio. This example uses `estimateCustomObjectivePortfolio` to solve a problem for a minimum variance `Portfolio` with a tracking error penalty. The example also shows how to add the tracking error as a "soft constraint" to any portfolio optimization problem. The tracking error measures the standard deviation of the divergence between a portfolio's return and that of a benchmark.

Define the mean and covariance of the assets returns.

```
% Define assets mean.
mu = [ 0.05; 0.1; 0.12; 0.18 ];
% Define assets covariance.
Sigma = [ 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 ];
```

Simulate a timeseries of assets returns with mean `mu` and covariance `Sigma`. From this point onward, assume that the true mean and covariance are not known and they can only be estimated from the timetable data.

```
% Find all business days of the last two years.
tEnd = datetime('today');
bdates = busdays(tEnd-calyears(2),tEnd);
% Create a simulated timetable.
rng('default')
nScen = size(bdates,1);
assetsTT = array2timetable(mvnrnd(mu,Sigma,nScen),RowTimes=bdates);
```

Define a long-only, fully invested portfolio problem using `Portfolio`.

```
% Define a Portfolio object.
p = Portfolio;
p = estimateAssetMoments(p,assetsTT);
% Specify constraints for a long-only, fully invested portfolio.
p = setDefaultConstraints(p);
```

Assume that there is a market index that follows the maximum Sharpe ratio portfolio. Use `estimateMaxSharpeRatio` to create the benchmark portfolio.

```
% Create the benchmark portfolio.
bmkPort = estimateMaxSharpeRatio(p);
% Create the benchmark returns timetable.
marketTT = timetable(assetsTT.Time,assetsTT.Variables*bmkPort, ...
    VariableNames={'Benchmark'});
```

Minimize Tracking Error

You can use two methods to obtain the tracking error. The first method uses the variance of the excess return of the portfolio with respect to the benchmark. This method assumes that the benchmark prices or returns are given, but does not require knowledge of the weights of the assets associated to the benchmark. The second method requires you to decompose the benchmark into the different weights for each asset.

Tracking Error Using Benchmark Returns

To compute the tracking error, you first need to compute the excess return of each asset with respect to the benchmark.

```
% Compute the excess return.
excessReturn = assetsTT.Variables-marketTT.Variables;
```

Then, you can compute the excess returns covariance matrix $\Sigma_{\widehat{R}}$.

```
excessRetSigma = cov(excessReturn);
```

Compute the tracking error as the variance of the excess return of the portfolio

$$x^T \Sigma_{\widehat{R}} x\,.$$

For this example, you want to find the portfolio that minimizes the tracking error to the benchmark with, at most, half of the assets in the universe. Furthermore, if an asset is selected in the portfolio, at least 10% must be invested in that asset. This problem is as follows:

$$\begin{aligned} &\min_x \; x^T \Sigma_{\widehat{R}} x \\ &s.t.\; \sum_i x_i = 1, \\ &\quad \sum_i \#(x_i \neq 0) \leq 2, \\ &\quad x = 0 \text{ or } x \geq 0.1 \end{aligned}$$

To solve this problem, first add the constraints to the `Portfolio` object using the `setBudget`, `setMinMaxNumAssets`, and `setBounds` functions.

```
% Specify that the budget sums to one constraint.
p = setBudget(p,1,1);
% Specify the maximum number of assets as 15.
p = setMinMaxNumAssets(p,[],2);
% Specify the conditional bounds.
p = setBounds(p,0.1,[],BoundType="conditional");
```

Define a function handle for the objective function $x^T \Sigma_{\widehat{R}} x$.

```
% Define the objective function.
TEsquared1 = @(x) x'*excessRetSigma*x;
```

Use `estimateCustomObjectivePortfolio` to compute the solution to the problem.

```
% Solve the porfolio problem.
wMinTE1 = estimateCustomObjectivePortfolio(p,TEsquared1)
```

```
wMinTE1 = 4×1

    0.8392
         0
         0
    0.1608
```

Tracking Error Using Benchmark Weights

You can also define the tracking error as

$$(x - x_0)^T \Sigma (x - x_0).$$

In this formulation, Σ is the covariance matrix of the assets returns (not the excess returns) and this example assumes that the weight vector of the benchmark portfolio is known. Using the same constraints as the Tracking Error Using the Benchmark Returns on page 4-343 method, the problem is as follows:

$$\begin{aligned} &\min_x \ (x - x_0)^T \Sigma (x - x_0) \\ &s.t. \sum_i x_i = 1, \\ &\quad \sum_i \#(x_i \neq 0) \leq 2, \\ &\quad x = 0 \text{ or } x \geq 0.01 \end{aligned}$$

Since the constraints are the same, you need only to define a new objective function.

```
% Define the objective function.
TESquared2 = @(x) (x-bmkPort)'*p.AssetCovar*(x-bmkPort);
```

Use `estimateCustomObjectivePortfolio` to compute the solution to the problem.

```
% Solve the porfolio problem.
wMinTE2 = estimateCustomObjectivePortfolio(p,TESquared2)
```

```
wMinTE2 = 4×1

    0.8392
         0
         0
    0.1608
```

The difference in the allocation for the different methods is negligible within a numerical accuracy. This means that the solution to both problems is the same.

```
norm(wMinTE1-wMinTE2,"inf")
```

```
ans = 0
```

Add Tracking Error Constraints

When you have a portfolio problem with cardinality constraints or conditional bounds and you use an objective function that is different than the return or variance for a mixed-integer problem, this portfolio problem does not support tracking error constraints. One option for including a tracking error constraint with a mixed-integer problem is to add the tracking error as a penalty to the objective function. In this case, the tracking error becomes a soft constraint and the strength of the constraint is controlled with the penalty parameter $\lambda \geq 0$. The larger λ is, the stronger the tracking error constraint becomes, and conversely, the smaller λ is, the weaker the tracking error constraint becomes.

Assume that you are interested in obtaining the portfolio closest to the equally weighted portfolio (EWP) such that the tracking error is smaller than 5.3%. Minimizing the Herfindahl-Hirschman (HH) index, given by x^Tx, returns the portfolio closest to the EWP that satisfies the necessary constraints. Assuming that the portfolio must satisfy the same constraints as in Minimizing Tracking Error on page 4-342, the problem is as follows:

$$\begin{aligned}
&\min_x \; x^T x + \lambda (x - x_0)^T \Sigma (x - x_0)\\
&s.t. \sum_i x_i = 1,\\
&\qquad \sum_i \#(x_i \neq 0) \leq 2,\\
&\qquad x = 0 \text{ or } x \geq 0.01
\end{aligned}$$

Define the penalized objective function. In this case, you can use the minimum tracking error portfolio as the benchmark portfolio.

```
% Define the objective function.
lambda = 100 ;
penalizedObjFun = @(x) x'*x + ...
    lambda*(x-bmkPort)'*p.AssetCovar*(x-bmkPort);
```

Use `estimateCustomObjectivePortfolio` to solve problem.

```
% Solve the penalized problem.
wEWPwithTE = estimateCustomObjectivePortfolio(p,penalizedObjFun)
```

```
wEWPwithTE = 4×1

    0.6312
    0.3688
         0
         0
```

Check if the tracking error satisfies the constraint. If it does not, make λ larger.

```
% Check if tracking error is smaller than 5.3%.
sqrt(TESquared2(wEWPwithTE)) <= 0.053
```

```
ans = logical
   1
```

Compare the different portfolio allocations.

```
% Compare the weights of the different portfolios.
pwgt = table(wEWPwithTE,wMinTE1,bmkPort, ...
    VariableNames={'EWPwithTE','MinTE','Benchmark'})
```

```
pwgt=4×3 table
    EWPwithTE     MinTE      Benchmark
    _________    _______    _________

     0.63115     0.83924     0.64539
     0.36885           0     0.14658
```

```
         0           0    0.088365
         0     0.16076     0.11967
```

When you compare the weights of the portfolio that minimizes the HH index with the tracking error penalty against the weights of the portfolio that minimizes the tracking error, you can see:

- As $\lambda \to 0$, the weights of the penalized problem become exactly 0.5 in two assets. This happens because a portfolio with 0.5 weights in two assets is the one that is closest to the EW portfolio and satisfies the constraints.
- As $\lambda \to \infty$, the weights of the penalized problem come closer to the weights of the minimum tracking error problem. This result happens because the penalized term in the objective overcomes the HH index.

Selecting different values of $\lambda \geq 0$ modifies the strength of the tracking error constraint.

See Also

`estimatePortSharpeRatio` | `estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierByRisk` | `estimateCustomObjectivePortfolio`

Related Examples

- “Diversify Portfolios Using Custom Objective” on page 4-329
- “Portfolio Optimization Using Social Performance Measure” on page 4-257
- “Diversify Portfolios Using Custom Objective” on page 4-329
- “Portfolio Optimization Against a Benchmark” on page 4-195
- “Risk Parity or Budgeting with Constraints” on page 4-356
- “Solve Robust Portfolio Maximum Return Problem with Ellipsoidal Uncertainty” on page 4-349
- “Solve Problem for Minimum Tracking Error with Net Return Constraint” on page 4-347

More About

- “Solver Guidelines for Custom Objective Problems Using Portfolio Objects” on page 4-115

Solve Problem for Minimum Tracking Error with Net Return Constraint

This example shows how to use `estimateCustomObjectivePortfolio` to solve a portfolio problem for minimum tracking error with a net return constraint using a custom objective.

Create `Portfolio` Object

Create a `Portfolio` object.

```
% Create a Portfolio object
load('SixStocks.mat')
p = Portfolio(AssetMean=AssetMean,AssetCovar=AssetCovar);
```

Define Problem

The portfolio problem for a minimum tracking error problem with a net return constraint is defined as

$$\min_{x}\ (x-x_0)^T\Sigma(x-x_0)$$
$$s.t.\, \mu^T x - c_B^T \max(0, x-x_0) - c_S^T \max(0, x_0 - x) \geq \mu_0$$
$$\sum_i x_i = 1$$
$$x \geq 0$$

Define Problem Parameters

Define the problem parameters for the `Portfolio` object.

```
% Initial portfolio
initPort = 1/p.NumAssets*ones(p.NumAssets,1);
% Buy cost
buyCost = 0.001*ones(p.NumAssets,1);
% Sell cost
sellCost = 0.002*ones(p.NumAssets,1);
% Net return target
ret0 = 0.03;
```

Solve Portfolio Problem

Use `estimateCustomObjectivePortfolio` to solve this portfolio problem for a minimum tracking error with a net return constraint. When using the `estimateCustomObjectivePortfolio` function with a `Portfolio` object, you add return constraints by using the `estimateCustomObjectivePortfolio` name-value argument `TargetReturn` with a return target value.

```
% Long-only, fully invested portfolio
p = setDefaultConstraints(p);
% Set the buy and sell costs
p = setCosts(p,buyCost,sellCost,initPort);

% Set the objective
fun = @(x) (x-initPort)'*p.AssetCovar*(x-initPort);
```

```
% Solve the portfolio problem
wFinTbx = estimateCustomObjectivePortfolio(p,fun,TargetReturn=ret0)

wFinTbx = 6×1

    0.1633
    0.0648
    0.1950
    0.2618
    0.0625
    0.2525
```

See Also

`estimatePortSharpeRatio` | `estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierByRisk` | `estimateCustomObjectivePortfolio`

Related Examples

- “Diversify Portfolios Using Custom Objective” on page 4-329
- “Portfolio Optimization Using Social Performance Measure” on page 4-257
- “Diversify Portfolios Using Custom Objective” on page 4-329
- “Portfolio Optimization Against a Benchmark” on page 4-195
- “Solve Problem for Minimum Variance Portfolio with Tracking Error Penalty” on page 4-342
- “Solve Robust Portfolio Maximum Return Problem with Ellipsoidal Uncertainty” on page 4-349
- “Risk Parity or Budgeting with Constraints” on page 4-356

More About

- “Solver Guidelines for Custom Objective Problems Using Portfolio Objects” on page 4-115

Solve Robust Portfolio Maximum Return Problem with Ellipsoidal Uncertainty

This example shows a robust formulation of portfolio optimization with uncertainty in the assets returns. It uses `estimateCustomObjectivePortfolio` to solve a robust `Portfolio` problem for a maximum return problem with an ellipsoidal uncertainty. Robust portfolio optimization, in contrast to the deterministic Markowitz mean-variance formulation, considers the uncertainty in the parameters of the problem: the assets' expected returns and their covariance matrix. In the traditional mean-variance model, the mean and covariance are assumed to take the point-values of the historical mean and covariance. However, in robust optimization, the mean and covariance are in a set that contains the most likely realizations, and the problem is optimized with respect to the worst possible outcome in that set.

Define Problem

If the vector of expected returns r has no uncertainty, the portfolio optimization problem that maximizes the returns is

$$R_{\text{port}} = \max_{x \in X} r^T x,$$

where

- $x \in \mathbb{R}^n$ is the vector of portfolio weights.
- X is the set of feasible allocations represented by a set of constraints.

Robust optimization assumes that the estimates of the expected returns are unreliable, but live in a set that contains the most likely realizations. The set of possible realizations of the expected returns is the *uncertainty set* and is given by

$$\{r \mid r \in \mathrm{S}(r_0)\},$$

where

- $\mathrm{S}(r_0)$ is a region around the vector r_0.

A common robust formulation of the portfolio problem is the one that tries to maximize the expected return of the portfolio in the worst case scenario of the uncertainty set. This is represented with the following "max-min" problem

$$R_{\text{port}} = \max_{x \in X} \min_{r \in \mathrm{S}(r_0)} r^T x .$$

In this example, assume that the uncertainty set follows the common ellipsoidal uncertainty given by

$$\mathrm{S}(r_0) = \left\{ r \mid (r - r_0)^T \Sigma_r^{-1} (r - r_0) \le \kappa^2 \right\},$$

where

- κ is the uncertainty aversion parameter that defines the width of the uncertainty.
- Σ_r is the covariance matrix of estimation errors in the expected returns r.

Goldfarb and Iyengar [1 on page 4-353] show that the robust maximization return problem with ellipsoidal uncertainty in the return is formulated as

$$\max_{x \in X} \; r_0^T x - \kappa\sqrt{x^T \Sigma_r x}\,.$$

Build Matrix of Estimated Errors in Expected Returns

Assume that the covariance matrix of estimation errors Σ_r is a diagonal matrix whose entries are proportional to the assets variance.

Load the data.

```
% Read the table of daily adjusted close prices for 2006 DJI stocks.
T = readtable('dowPortfolio.xlsx');
% Convert the table to a timetable.
pricesTT = table2timetable(T);
% Remove the DJI stock from the table.
pricesTT = pricesTT(:,2:end);
```

Build the covariance matrix of estimation errors in the expected returns.

```
% Compute the returns from the prices.
returnsTT = tick2ret(pricesTT);
% Compute the assets covariance matrix.
Sigma = cov(returnsTT.Variables);
% Compute the covariance matrix of estimation errors in the expected returns.
SigmaR = diag(diag(Sigma));
```

Solve Robust Problem

Define a `Portfolio` object.

```
% Define a traditional mean-variance portfolio.
p = Portfolio;
p = estimateAssetMoments(p,returnsTT,MissingData=true,GetAssetList=true);
```

Add constraints to the `Portfolio` object. The portfolio is a fully invested, long-short portfolio with bounds `[-0.2,0.2]`.

```
% Specify a fully invested constraint.
p = setBudget(p,1,1);
% Specify the long-short bounds.
p = setBounds(p,-0.2,0.2);
```

Define the objective function with $\kappa = 0.5$ as the ellipsoidal uncertainty parameter.

```
% Objective function handle
kappa = 0.5;
robustObjective = @(x) p.AssetMean'*x - kappa*sqrt(x'*SigmaR*x);
```

Solve the robust maximum return problem using `estimateCustomObjectivePortfolio`.

```
wRobustMaxRet = estimateCustomObjectivePortfolio(p,robustObjective,ObjectiveSense="maximize");
```

Compare Mean-Variance and Robust Portfolio Strategies

Compare the performance of the traditional maximum return portfolio against its robust counterpart using backtesting.

Define the warmup period to compute the initial portfolios for the different strategies using the data in that period.

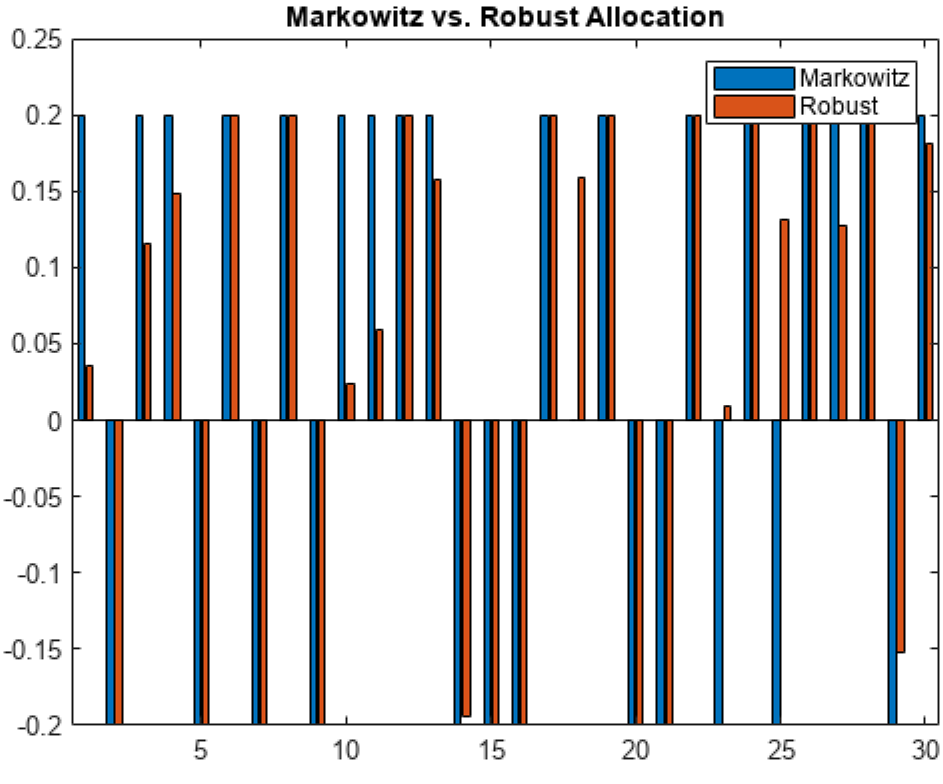
```
% Set backtesting warmup period to two 21-day months.
warmupPeriod = 21*2;
% Set a warmup partition of the timetable.
warmupTT = pricesTT(1:warmupPeriod,:);
```

Compute the initial portfolios using the rebalancing functions (`markowitzFcn` and `robustFcn`) that are defined in Local Functions on page 4-354.

```
% Define the initial Markowitz portfolio.
initialMarkowitz = markowitzFcn(zeros(p.NumAssets,1),warmupTT,p);
% Define the initial robust portfolio.
initialRP = robustFcn(zeros(p.NumAssets,1),warmupTT,p,kappa);
```

Plot both types of portfolios to compare the initial behavior.

```
% Plot the intial portfolios.
bar([initialMarkowitz initialRP])
legend('Markowitz','Robust')
title('Markowitz vs. Robust Allocation')
```



Out of 30 assets in the Markowitz allocation, 29 are at the extremes of the feasible bounds. On the other hand, only 17 assets in the robust allocation are at their extremes. The number of assets at the extremes of the feasible bounds shows the sensitivity of the traditional maximum return allocation strategy to the parameters of the problem. The Markowitz strategy invests everything possible in the assets with the highest returns. This result means that the Markowitz strategy shorts the assets with the lowest returns to be able to allocate more to the assets with the largest returns. On the other hand, the robust strategy does not have the same extreme behavior.

Define Backtesting Parameters

Set the backtesting strategies to rebalance every month.

```
% Define the monthly rebalance frequency.
rebalFreq = 21;
```

To gather enough data, set the `backtestStrategy` lookback window for the backtesting to at least 2 months. To remove old data from the parameter estimation, set the lookback window to no more than 6 months.

```
% Define the lookback window.
lookback = [42 126];
```

Use a fixed transaction cost equal to 0.5% of the amount traded.

```
% Define a fixed transaction cost.
transactionCost = 0.005;
```

Define the allocation strategies using `backtestStrategy`.

```
% Define a traditional mean-variance strategy.
stratMarkowitz = backtestStrategy('Markowitz', @(w,TT) markowitzFcn(w,TT,p), ...
    RebalanceFrequency=rebalFreq, ...
    LookbackWindow=lookback, ...
    TransactionCosts=transactionCost, ...
    InitialWeights=initialMarkowitz);

% Define a robust strategy.
stratRobust = backtestStrategy('Robust', @(w,TT) robustFcn(w,TT,p,kappa), ...
    RebalanceFrequency=rebalFreq, ...
    LookbackWindow=lookback, ...
    TransactionCosts=transactionCost, ...
    InitialWeights=initialRP);
```

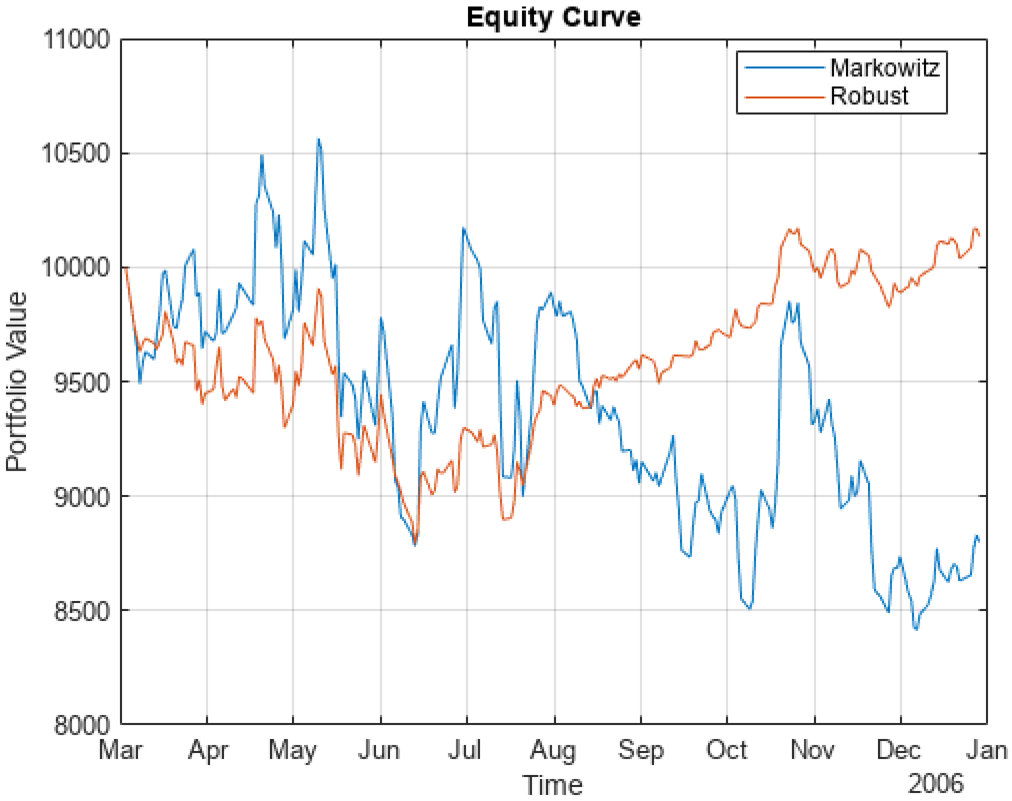
Run Backtest

Create a `backtestEngine` object and then run the backtest using `runBacktest`.

```
% Define strategies for backtest engine
strategies = [stratMarkowitz stratRobust];
% Define a backtest engine
backtester = backtestEngine(strategies);
% Run the backtest
backtester = runBacktest(backtester,pricesTT,Start=warmupPeriod);
```

Use `equityCurve` to plot the equity curve.

```
equityCurve(backtester)
```



Use `summary` to generate a table of performance results for both strategies.

```
summary(backtester)
```

```
ans=9×2 table
                          Markowitz       Robust
                         ___________    __________

    TotalReturn             -0.12004      0.013641
    SharpeRatio             -0.02926      0.011774
    Volatility              0.016383     0.0088013
    AverageTurnover         0.047756      0.021642
    MaxTurnover               1.8849        1.0392
    AverageReturn        -0.00047822    0.00010338
    MaxDrawdown              0.20332       0.12068
    AverageBuyCost             2.276         1.036
    AverageSellCost            2.276         1.036
```

As expected, the robust strategy has lower volatility than the traditional mean-variance strategy. This characteristic is observed in all the performance indicators that measure variability: volatility, turnover, and maximum drawdown. This result is the expected outcome of robust strategies. Although it is not always the case, in this particular example, the robust strategy also outperforms the traditional mean-variance allocation.

References

[1] Goldfarb, D. and G. Iyengar. "Robust Portfolio Selection Problems." *Mathematics of Operations Research*. *28*(1), pp. 1–38, 2003.

Local Functions

```
function new_weights = markowitzFcn(~,pricesTT,portObj)
% Traditional mean-variance portfolio maximum return allocation

% Estimate the portfolio's mean and variance.
p = estimateAssetMoments(portObj,pricesTT,DataFormat='Prices', ...
    MissingData=true);

% Compute the max return portfolio.
new_weights = estimateFrontierLimits(p,'max');

end

function new_weights = robustFcn(~,pricesTT,portObj, ...
    kappa)
% Robust portfolio maximum return allocation

% Estimate the portfolio's mean and variance.
p = estimateAssetMoments(portObj,pricesTT,DataFormat='Prices', ...
    MissingData=true);

% Cumpute covariance matrix of estimation errors in the expected returns.
SigmaR = diag(diag(p.AssetCovar));

% Define the objective function handle.
robustObjective = @(x) p.AssetMean'*x - kappa*sqrt(x'*SigmaR*x);

% Compute the maximum return portfolio.
new_weights = estimateCustomObjectivePortfolio(p,robustObjective, ...
    ObjectiveSense="maximize");

end
```

See Also

estimatePortSharpeRatio | estimateFrontier | estimateFrontierByReturn | estimateFrontierByRisk | estimateCustomObjectivePortfolio

Related Examples

- “Diversify Portfolios Using Custom Objective” on page 4-329
- “Portfolio Optimization Using Social Performance Measure” on page 4-257
- “Diversify Portfolios Using Custom Objective” on page 4-329
- “Portfolio Optimization Against a Benchmark” on page 4-195
- “Solve Problem for Minimum Variance Portfolio with Tracking Error Penalty” on page 4-342
- “Solve Problem for Minimum Tracking Error with Net Return Constraint” on page 4-347
- “Risk Parity or Budgeting with Constraints” on page 4-356

More About

- “Solver Guidelines for Custom Objective Problems Using Portfolio Objects” on page 4-115

Risk Parity or Budgeting with Constraints

This example shows how to solve risk parity or budgeting problems with constraints using `estimateCustomObjectivePortfolio`.

Risk parity is a portfolio allocation strategy that focuses on the allocation of risk to define the weights of a portfolio. You construct the risk parity portfolio, or equal risk contribution portfolio, by ensuring that all assets in a portfolio have the same risk contribution to the overall portfolio risk. The generalization of this problem is the risk budgeting portfolio in which you set the risk contribution of each asset to the overall portfolio risk. In other words, instead of all assets contributing equally to the risk of the portfolio, the assets' risk contribution must match a target risk budget.

In “Risk Budgeting Portfolio” on page 4-280, the example describes some advantages of risk parity or budgeting portfolios. It shows how to use `riskBudgetingPortfolio` to solve long-only, fully invested risk parity or budgeting problems. When the only constraints of the risk budgeting problem enforce nonnegative weights that sum to 1, the risk budgeting problem is a feasibility problem. In other words, the solution to the problem is the one that exactly matches the target risk budget and it is unique. Therefore, there is no need for an objective function. The `riskBudgetingPortfolio` function is optimized to solve this specific type of problems.

When you add more constraints to the risk parity problem, you must find the weight allocation that satisfies the extra constraints while trying to match the target risk budget as much as possible. The next section explains how.

Define Problem

Start by defining the problem data.

```
% Assets variance
vol = [0.1;0.15;0.2;0.3];
% Assets correlation
rho = [1.00  0.50  0.50  0.50;
       0.50  1.00  0.50  0.50;
       0.50  0.50  1.00  0.75;
       0.50  0.50  0.75  1.00];
% Assets covariance matrix
Sigma = corr2cov(vol,rho);
% Risk budget
budget = [0.3;0.3;0.195;0.205];
```

Obtain the long-only, fully invested risk budgeting portfolio to use as benchmark.

```
% Long-only fully invested risk budgeting portfolio
w_simple = riskBudgetingPortfolio(Sigma,budget);
% Compute the risk contribution of the weight allocation
RC_simple = portfolioRiskContribution(w_simple,Sigma);
% Table with weights and risk contributions
T_simple = table(w_simple,RC_simple,budget, ...
    VariableNames={'RB Portfolio','RB Contribution','Budget'})
```

```
T_simple=4×3 table
    RB Portfolio    RB Contribution    Budget
    ____________    _______________    ______

      0.45053             0.3            0.3
    0.30035              0.3           0.3
    0.14668            0.195         0.195
    0.10244            0.205         0.205
```

As desired, the risk contribution of the risk budgeting portfolio matches the `budget`.

Add Constraints to Risk Parity or Budgeting Portfolio

When you add constraints, other than the long-only, fully invested constraints, to the risk parity formulation, you must find the portfolio allocation that satisfies all the constraints and minimizes the deviation from the target risk budget.

The percent risk contribution of asset i is defined as

$$\mathrm{RC}_i(w) = \frac{w_i(\Sigma w)_i}{w^T\Sigma w},$$

and the goal is to find the portfolio that minimizes the deviation of $\mathrm{RC}_i(w)$ to b_i, the target risk budget of asset i. For risk parity problems, $b_i = \frac{1}{n}$, where n is the number of assets.

A common way to measure deviation is to use a norm. In this example, use the 2-norm. The risk parity or budgeting problem with constraints results in

$$\begin{aligned} &\min_w \sum_i \left(\frac{w_i(\Sigma w)_i}{w^T\Sigma w} - b_i\right)^2 \\ &s.t.\ \ w \in \Omega \end{aligned}$$

where Ω represents the feasible set that is defined by the desired constraints. The objective function is the sum of squares of the deviation of the risk contributions from the target. Notice that the objective function is nonconvex.

Next, introduce the constraint $x_i \le 0.3$, which means that no more than 30% of the capital can be invested in each asset.

```
% Define the Portfolio object
p = Portfolio(AssetCovar=Sigma);
% Set bound constraints
p = setBounds(p,0,0.3);
% Set budget constraint
p = setBudget(p,1,1);
```

Create a function handle for the nonconvex objective

$$\sum_i \left(\frac{w_i(\Sigma w)_i}{w^T\Sigma w} - b_i\right)^2.$$

```
% Define objective function
objFun = @(w) sum(((w.*(Sigma*w))/(w'*Sigma*w) - budget).^2);
```

Solve the problem using `estimateCustomObjectivePortfolio` and `riskBudgetingPortfolio`.

```
% Risk budgeting portfolio with extra constraints
w_extra = estimateCustomObjectivePortfolio(p,objFun);
% Risk contribution
```

```
RC_extra = portfolioRiskContribution(w_extra,Sigma);
% Table with weights and risk contributions
T_extra = table(w_extra,RC_extra,budget, ...
    VariableNames={'RB Portfolio','RB Contribution','Budget'})
```

```
T_extra=4×3 table
    RB Portfolio    RB Contribution    Budget
    ____________    _______________    ______

        0.3             0.15505          0.3 
        0.3             0.24998          0.3 
     0.2465             0.30862        0.195 
     0.1535             0.28635        0.205 
```

Now that you have added additional constraints, the risk contribution of the resulting portfolio does not match the budget. Yet, the solution to the problem is minimizing the deviation of the risk contribution to the target risk budget. Because the risk budgeting portfolio without the extra constraints assigns weights larger than 30% to the first and second assets, the solution to the problem that minimizes the deviation assigns as much as possible to the first and second assets and distributes the rest to the other assets.

As explained in “Role of Convexity in Portfolio Problems” on page 4-148, because the objective function is nonconvex, this formulation cannot be solved by the `Portfolio` object solvers if you add cardinality constraints or conditional bounds to the problem.

Compare Risk Parity and Mean-Variance Portfolios

You can examine the difference in the allocation between the traditional minimum variance portfolio and the risk parity portfolio.

First compute the minimum variance portfolio. The traditional mean-variance framework requires the assets mean return to be defined before you compute any of the portfolios on the efficient frontier.

```
% Define assets mean
p.AssetMean = [0.1; 0.1; 0.15; 0.2];
% Minimum variance portfolio
wMinVar = estimateFrontierLimits(p,'min');
```

To compute the risk parity portfolio, make sure all assets contribute equally to the risk. Set $b_i = \frac{1}{n}$ and update the objective function.

```
% Update budget
budget = 1/p.NumAssets*ones(p.NumAssets,1);
% Update objective
objFun = @(w) sum(((w.*(Sigma*w))/(w'*Sigma*w) - budget).^2);
```

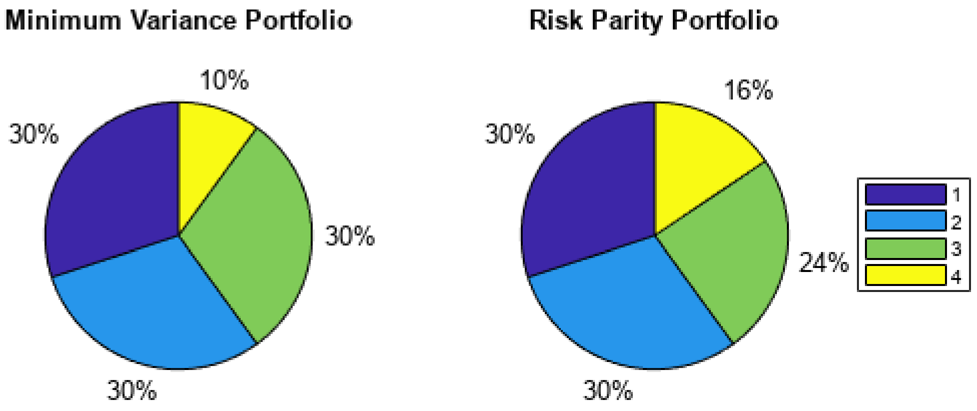
Solve the risk parity problem using `estimateCustomObjectivePortfolio`.

```
% Risk parity portfolio with extra constraints
wRP = estimateCustomObjectivePortfolio(p,objFun);
```

Compare the weight allocation concentrations.

```
% Plot pie charts
tiledlayout(1,2);
% Minimum variance portfolio
```

```
nexttile
pie(wMinVar)
title('Minimum Variance Portfolio',Position=[0,1.5]);
% Risk parity portfolio
nexttile
pie(wRP)
title('Risk Parity Portfolio',Position=[0,1.5]);

% Add legend
lgd = legend({'1','2','3','4'});
lgd.Layout.Tile = 'east';
```



The portfolio allocation of the minimum variance portfolio sets all assets with the smallest risk to their maximum (30%). On the other hand, the risk parity allocation sets only the first two assets to 30% and the last two are more balanced. This result is a simple example of how the risk parity allocation helps to diversify a portfolio.

See Also

`estimatePortSharpeRatio` | `estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierByRisk` | `estimateCustomObjectivePortfolio`

Related Examples

- “Diversify Portfolios Using Custom Objective” on page 4-329
- “Portfolio Optimization Using Social Performance Measure” on page 4-257

- “Diversify Portfolios Using Custom Objective” on page 4-329
- “Portfolio Optimization Against a Benchmark” on page 4-195
- “Solve Problem for Minimum Variance Portfolio with Tracking Error Penalty” on page 4-342
- “Solve Problem for Minimum Tracking Error with Net Return Constraint” on page 4-347
- “Solve Robust Portfolio Maximum Return Problem with Ellipsoidal Uncertainty” on page 4-349

More About

- “Solver Guidelines for Custom Objective Problems Using Portfolio Objects” on page 4-115

Single Period Goal-Based Wealth Management

This example shows a method for goal-based wealth management (GBWM). In GBWM, risk is not necessarily measured using the standard deviation, the value-at-risk, or any other common risk measure. Instead, risk is understood as the likelihood of not attaining an investor's goal. You choose a weight allocation that is on the traditional mean-variance efficient frontier and that also maximizes the probability of exceeding a wealth goal at the end of the investment horizon. In other words, you choose the portfolio on the efficient frontier that minimizes the risk of not attaining the investor's goal.

A common issue after computing the mean-variance efficient frontier is how to choose a single portfolio to invest in. This portfolio could be the maximum Sharpe ratio portfolio, which selects the weights that achieve the highest return given the amount of risk taken. However, choosing this portfolio does not take the investor's preferences into account. Another way strategy is to gauge the risk preference of the investor and select a portfolio that achieves a risk no larger than the investor's preferred risk. However, it is difficult to quantify the risk aversion of an investor. In this example, you use the probability of achieving a wealth goal to help choose a portfolio on the efficient frontier.

Load Data

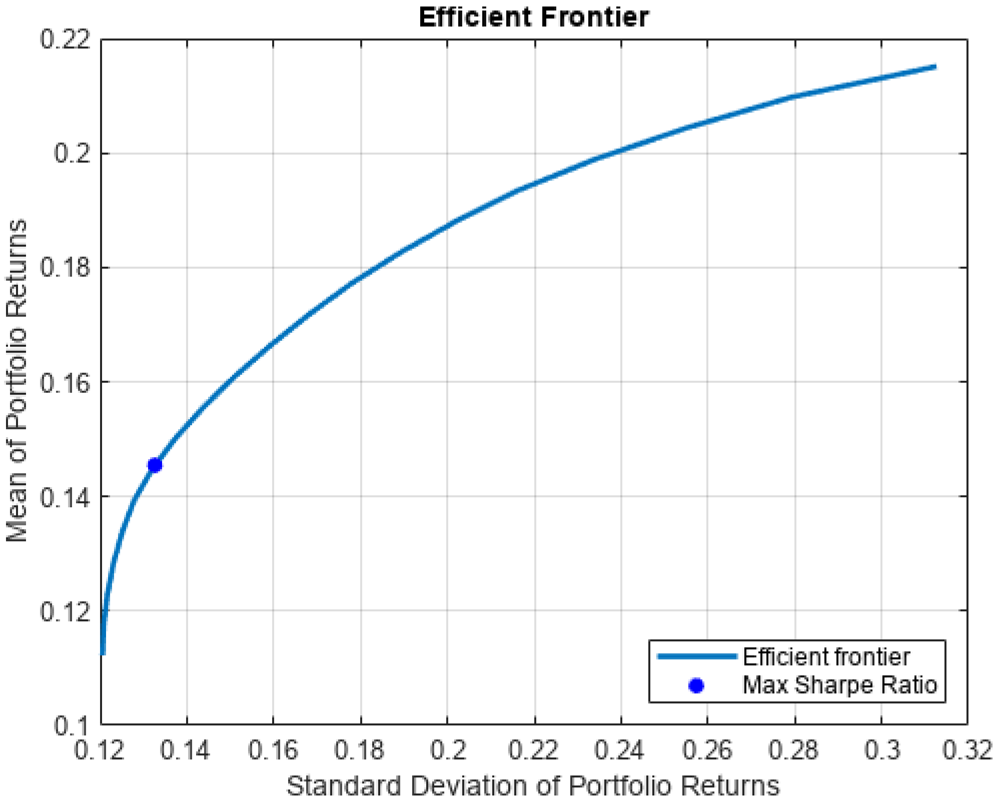
Load the data and then annualize the asset returns and the variance-covariance matrix.

```
% Load data
load BlueChipStockMoments.mat
AssetMean = 12*AssetMean;
AssetCovar = 12*AssetCovar;
```

Plot Efficient Frontier

Use `Portfolio` to create a `Portfolio` object and `estimateMaxSharpeRatio` to compute the maximum Sharpe ratio. Then, use `plotFrontier` to plot the efficient frontier for long-only, fully-invested portfolios and mark the maximum Sharpe ratio portfolio.

```
% Create Portfolio object
p = Portfolio(AssetList=AssetList,AssetMean=AssetMean,...
    AssetCovar=AssetCovar);

% Add long-only, fully-invested constraints
p = setDefaultConstraints(p);

% Compute maximum Sharpe ratio portfolio
wSR = estimateMaxSharpeRatio(p);

% Plot efficient frontier
numPorts = 20;
figure;
[pret,prisk] = plotFrontier(p,20);
hold on
plot(estimatePortRisk(p,wSR),estimatePortReturn(p,wSR),'b.',...
    'MarkerSize',20);
legend('Efficient frontier','Max Sharpe Ratio',...
    'Location','southeast')
hold off
```



Choose Single Portfolio Using GBWM

GBWM provides a way to choose a single portfolio on the efficient frontier that allows you to incorporate the investor's preference in a straightforward way. In this case, the investor's goal is to reach a target wealth G from their initial wealth W_0 at the end of the investment period T. Therefore, you can formulate the problem as choosing the portfolio that maximizes the probability of having a wealth at time T that surpasses the goal G, while making sure that the chosen weights are on the efficient frontier.

$$\max_{w,\mu,\sigma} \; P(W_T \geq G)$$

$$s.t. \;\; {\mu_0}^T w = \mu$$

$$w^T \Sigma_0 w = \sigma^2$$

$$\sum_{i=1}^{n} w_i = 1$$

$$0 \leq w \leq 1$$

```
% Define investor's preferences
W0 = 100;
G = 200;
T = 10;
```

For this example, assume that the wealth evolves following a Brownian motion [1 on page 4-365]

$$W_T = W_0 \exp\left\{\left(\mu - \frac{\sigma^2}{2}\right)T + \sigma\sqrt{T}Z\right\},$$

where Z is a standard normal random variable.

The approach in this example also assumes that the returns follow a stationary distribution. In other words, the approach assumes that the returns have the same expectation and covariance throughout the investment period. The term “single-period” refers to the fact that the optimal allocation is computed *only once* at the beginning of the investment horizon assuming that the allocations will remain constant. This is a “set-it-and-forget-it” strategy. Although you can recompute the allocation weights at each reinvestment period by updating the time horizon and initial wealth, a single-period problem does not consider possible allocation changes in the middle of the investment horizon. In contrast, the optimal solution for a "multi-period" approach takes into account possible changes to the weights at intermediate investment periods. Thus, instead of just selecting the initial portfolio weights, multi-period problems compute the optimal strategy that should be followed given the changes in wealth at each of the reinvestment periods. The “Dynamic Portfolio Allocation in Goal-Based Wealth Management for Multiple Time Periods” on page 4-366 example shows a multi-period version of the problem in this example.

Define Objective Function and Solve Problem

Find the portfolio that maximizes the probability of achieving the investor's goal given the prior assumptions. Maximizing $P(W_T \geq G)$ is equivalent to minimizing $P(W_T \leq G)$, so the objective function is rewritten as

$$\max_{w,\mu,\sigma} P(W_T \geq G) = \min_{w,\mu,\sigma} P(W_T \leq G) = \min_{w,\mu,\sigma} \Phi\left(\frac{1}{\sigma\sqrt{T}}\left[\ln\left(\frac{G}{W_0}\right) - \left(\mu - \frac{\sigma^2}{2}\right)T\right]\right).$$

Since the relationship of μ and σ to the weights w is represented by equality constraints in the problem, you can substitute these values in the objective function and remove the equality constraints that show the relationship between these three variables.

$$\max_{w,\mu,\sigma} P(W_T \geq G) = \min_{w} \Phi\left(\frac{1}{\sqrt{T\left(w^T\Sigma_0 w\right)}}\left[\ln\left(\frac{G}{W_0}\right) - \left(\mu_0^T w - \frac{w^T\Sigma_0 w}{2}\right)T\right]\right)$$

Define the objective function handle using the definition above and then use `estimateCustomObjectivePortfolio` to solve the problem.

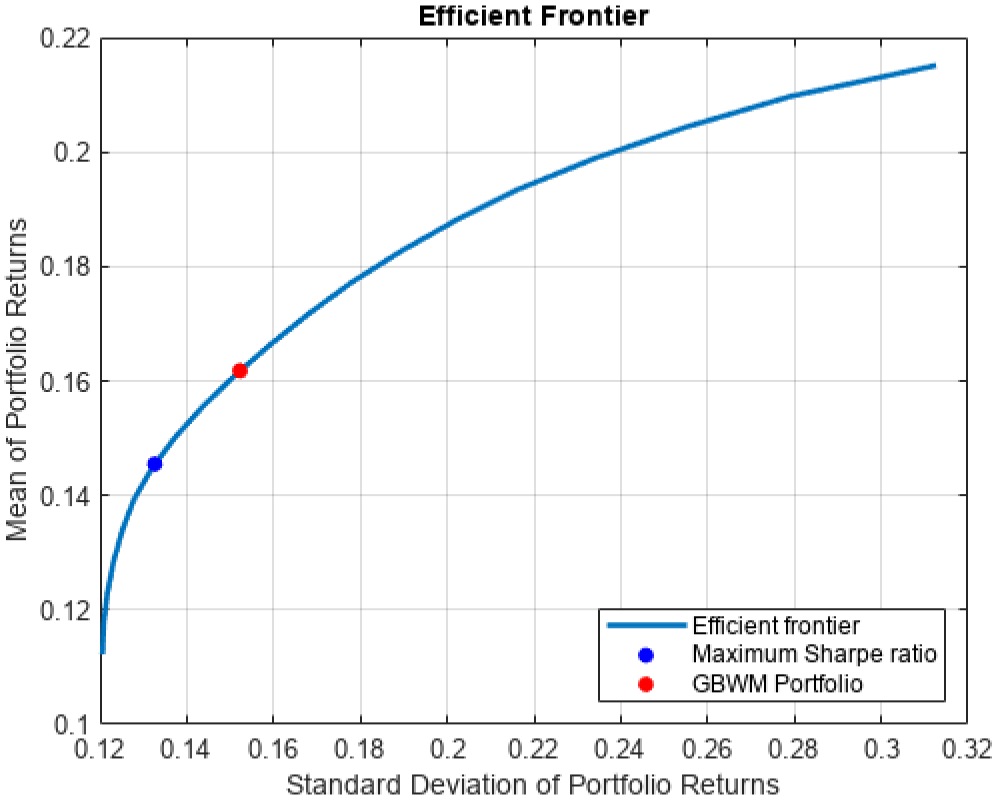
```
% Define objective function
probability = @(w) normcdf(1/sqrt(T*(w'*p.AssetCovar*w)) * ...
    (log(G/W0) - (p.AssetMean'*w - (w'*p.AssetCovar*w)/2)*T));

% Solve problem
wGBWM = estimateCustomObjectivePortfolio(p,probability);
```

Plot Efficient Frontier

Use `plotFrontier` to plot the efficient frontier, the maximum Sharpe ratio portfolio, and the portfolio that maximizes the probability of achieving the goal (called the GBWM portfolio).

```
% Plot efficient frontier
figure;
plotFrontier(p,pret,prisk);
hold on
```

```
plot(estimatePortRisk(p,wSR),estimatePortReturn(p,wSR),'b.', ...
    'MarkerSize',20);
plot(estimatePortRisk(p,wGBWM),estimatePortReturn(p,wGBWM),'r.', ...
    'MarkerSize',20);
legend('Efficient frontier','Maximum Sharpe ratio', ...
    'GBWM Portfolio','Location','southeast')
hold off
```



The GBWM portfolio is riskier than the maximum Sharpe ratio portfolio. This extra risk is necessary to maximize the probability of attaining the investor's wealth goal G by time T.

Compare GBWM Portfolio to Maximum Sharpe Ratio Portfolio

Compare the probabilities of acheiving the wealth goal G by the end of the investment period T for the maximum Sharpe ratio portfolio against the GBWM portfolio.

```
Tprobabilities = table(1-probability(wSR),1-probability(wGBWM),...
    RowNames={'P(W_T >= G)'},VariableNames={'SR','GBWM'})
```

```
Tprobabilities=1×2 table
                    SR         GBWM  
                 _______    _______

    P(W_T >= G)  0.94592    0.95347
```

The probability of successfully meeting the investor's wealth goal does not change much between the maximum Sharpe ratio portfolio and the GBWM portfolio. Using this information, an investor can understand the trade-off between acheiving their wealth goal G by time T compared to choosing a less risky portfolio. For example, an investor might choose a minimum variance portfolio that has a much lower probability of acheiving their wealth goal G by time T. Find the minimum variance (MV) portfolio, and compare the probability of achieving the wealth goal using this portfolio with the other two approaches.

```
wMV = estimateFrontierLimits(p,'min');
Tprobabilities.MV = 1-probability(wMV)
```

```
Tprobabilities=1×3 table
                     SR        GBWM        MV  
                   _______    _______    ______

    P(W_T >= G)    0.94592    0.95347    0.8252
```

References

[1] Das, Sanriv R., Daniel Ostrov, Anand Radhakrishnan, and Deep Srivastav. “A New Approach to Goals-Based Wealth Management.” *Journal of Investment Management.* 16, no. 3 (2018): 1–27.

See Also

`Portfolio` | `estimatePortSharpeRatio` | `estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierByRisk` | `estimateCustomObjectivePortfolio`

Related Examples

- “Dynamic Portfolio Allocation in Goal-Based Wealth Management for Multiple Time Periods” on page 4-366
- “Diversify Portfolios Using Custom Objective” on page 4-329
- “Portfolio Optimization Against a Benchmark” on page 4-195

More About

- “Solver Guidelines for Custom Objective Problems Using Portfolio Objects” on page 4-115

Dynamic Portfolio Allocation in Goal-Based Wealth Management for Multiple Time Periods

This example shows a dynamic programming strategy to maximize the probability of obtaining an investor's wealth goal at the end of the investment horizon. This dynamic programming strategy is known in the literature as a goal-based wealth management (GBWM) strategy. In GBWM, risk is not necessarily measured using the standard deviation, the value-at-risk, or any other common risk metric. Instead, risk is understood as the likelihood of not attaining an investor's goal. This alternative concept of risk implies that, sometimes, in order to increase the probability of attaining an investor’s goal, the optimal portfolio’s traditional risk (that is, standard deviation) must increase if the portfolio is underfunded. In other words, for the investor’s view of risk to *decrease*, the traditional view of risk must *increase* if the portfolio’s wealth is too low.

The objective of the investment strategy is to determine a dynamic portfolio allocation such that the portfolios chosen at each rebalancing period are on the traditional mean-variance efficient frontier and these portfolios maximize the probability of achieving the wealth goal G at the time horizon T. This objective is represented as

$$\max_{\{A(0), A(1), \ldots, A(T-1)\}} \mathbb{P}(W(T) \geq G),$$

where $W(T)$ is the terminal portfolio wealth and $A(t)$ are the possible actions and allocations at time $t = 0, 1, \ldots, T - 1$.

The dynamic portfolio problem consists of the following parts:

- Time periods — Portfolio rebalancing periods
- Actions — Portfolio weights rebalancing
- States — Wealth levels
- Value function — Probability of achieving the goal at the end of the investment period
- Policy or strategy — Optimal actions at each state and time period

This example follows the GBWM strategy of Das [1 on page 4-377]. For an example of GBWM for a single time period, see the “Single Period Goal-Based Wealth Management” on page 4-361 example.

Inputs for Investment Wealth Goals

Specify the initial wealth.

```
W0 = 100;
```

Specify the target wealth at the end of the investment horizon.

```
G = 200;
```

Define the time horizon.

```
T = 10;
```

Load the asset information.

```
load BlueChipStockMoments.mat
```

Rebalancing Period Actions for Efficient Portfolios

The action at each rebalancing period is to choose the weights of the investment portfolio for the next time period. In this example assume that the possible portfolio weights are assosicated with the portfolios on the efficient frontier. This example also assumes that the choice of possible portfolios is fixed throughout the full investment period. However, weights are not required to be on the efficient frontier. The only requirement is that the set of actions (portfolio choices) remains fixed throughout the investment period.

To simplify the computations, define a finite subset of the efficient portfolios for possible actions. Start by creating a `Portfolio` object with the asset information from `BlueChipStockMoments.mat`.

```
p = Portfolio(AssetList=AssetList,AssetMean=AssetMean,...
    AssetCovar=AssetCovar);
```

Annualize the asset returns and the variance-covariance matrix.

```
p.AssetMean  = 12*p.AssetMean;
p.AssetCovar = 12*p.AssetCovar;
```

Use `setBounds` to add constraints to the portfolio problem. Here, assets do not represent more than 10% of the portfolio wealth and shorting is not allowed.

```
p = setBounds(p,0,0.1);
```

Use `setBudget` to specify that the porfolio must be fully invested.

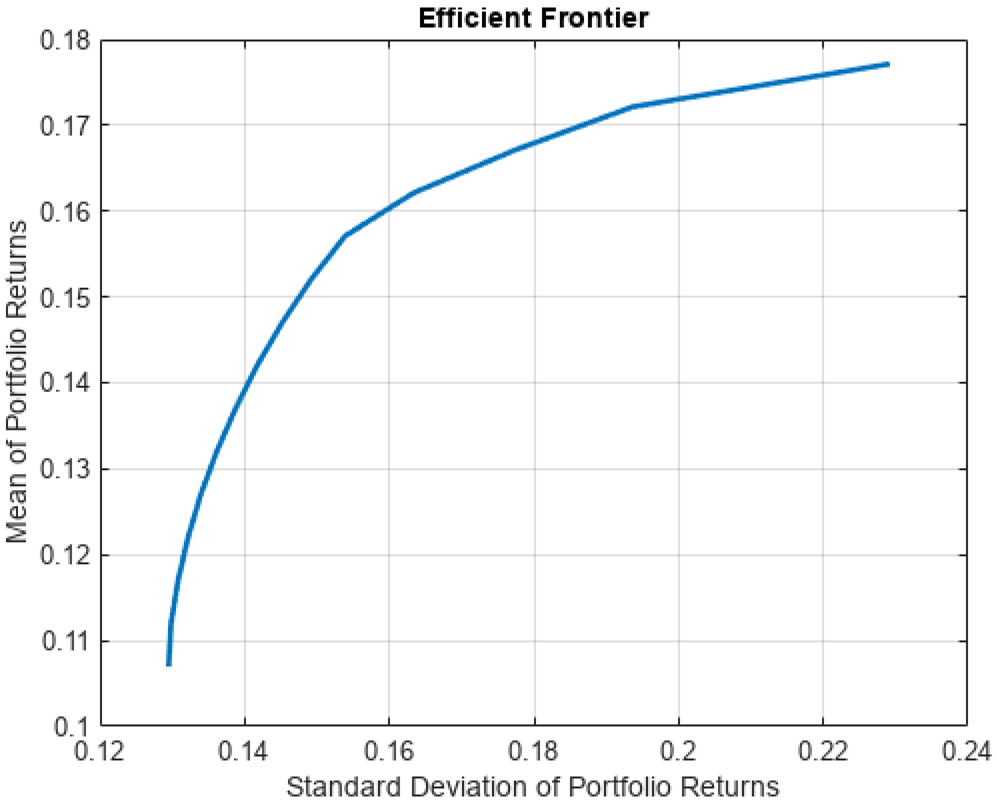
```
p = setBudget(p,1,1);
```

Compute the the weights of `15` portfolios on the efficient frontier using `estimateFrontier`. These portfolios represent the possible actions at each rebalancing period.

```
nPortfolios = 15;
pwgt = estimateFrontier(p,nPortfolios);
```

Plot the efficient frontier using `plotFrontier`.

```
[prsk,pret] = plotFrontier(p,pwgt);
```



States for Wealth Nodes

The states are defined as the wealth values at time t. To simplify the problem, discretize the wealth values to generate a wealth grid, W_i, for $i \in \{1, \ldots, i_{\max}\}.W_1$ corresponds to the smallest possible wealth $W_{\min}$ and $W_{i_{\max}}$ corresponds to the largest possible wealth $W_{\max}$. To determine $W_{\min}$ and $W_{\max}$, assume that the evolution of wealth follows a geometric Browninan motion

$$W(t) = W(0)e^{\left(\mu - \sigma^2/2\right)t + \sigma\sqrt{t}Z},$$

where Z is a standard normal random variable. GBWM does not require that the wealth evolution follow a Brownian motion. However, the technique in this example requires the evolution model to be Markovian. Therefore, this example assumes that the probability of being at a certain state at time $t + 1$ only depends on the state at time t and the action taken.

This example also assumes that Z realistically takes values between -3 and 3. When $Z = -3$, $W_{\min}$ corresponds to the smallest value obtained in the trajectory of the Brownian motion setting, μ is the smallest possible portfolio return $\mu_{\min}$, and σ is the largest possible portfolio risk $\sigma_{\max}$. Likewise, when $Z = 3$, $W_{\max}$ corresponds to the largest value in the trajectory, μ is the largest possible portfolio return $\mu_{\max}$, and σ is the largest possible portfolio risk $\sigma_{\max}$. For $W_{\max}$, use $\sigma_{\max}$ because the term in the exponential is increasing for all $\sigma \leq 3/\sqrt{t}$; if $\sigma_{\max} > 3/\sqrt{t}$, use $\sigma = 3/\sqrt{t}$ instead.

Define $\mu_{\min}$, $\mu_{\max}$, $\sigma_{\min}$, and $\sigma_{\max}$.

```
% Define return limits.
mu_min = pret(1);
```

```
mu_max = pret(end);

% Define volatility limits.
sigma_min = prsk(1);
sigma_max = prsk(end);
```

The smallest realistic value of W is

$$\widehat{W}_{\min} = \min_{\tau \in \{0, 1, \ldots, T\}} \left\{ W(0) e^{\left(\mu_{\min} - \frac{\sigma_{\max}^2}{2}\right)\tau - 3\sigma_{\max}\sqrt{\tau}} \right\}.$$

```
% Define the minimum possible wealth.
timeVec = 0:T;
WMinVec = W0*exp((mu_min-(sigma_max^2)/2)*timeVec- ...
    3*sigma_max*sqrt(timeVec));
WHatMin = min(WMinVec);
```

The largest realistic value of W is

$$\widehat{W}_{\max} = \max_{\tau \in \{0, 1, \ldots, T\}} \left\{ W(0) e^{\left(\mu_{\max} - \frac{\sigma_{\max}^2}{2}\right)\tau + 3\sigma_{\max}\sqrt{\tau}} \right\}.$$

```
% Define the maximum possible wealth.
WMaxVec = W0*exp((mu_max-(sigma_max^2)/2)*timeVec+ ...
    3*sigma_max*sqrt(timeVec));
WHatMax = max(WMaxVec);
```

Now, you can fill in the grid points between $\widehat{W}_{\min}$ and $\widehat{W}_{\max}$. When you set $Z = 1$and $t = 1$ and ignore the drift term $\left(\mu - \sigma^2/2\right)t$, note that σ is proportional to the logarithm of the wealth. Thus, you can work in logarithmic space and add a grid point every $\sigma_{\min}/\rho_{\text{grid}}$ units. ρ_{grid} is an exogenous parameter that determines the density of the grid. As ρ_{grid} grows, the number of wealth nodes increases.

```
% Transform to the logarithmic space.
lWHatMin = log(WHatMin);
lWHatMax = log(WHatMax);
% Define the grid in the logarithmic space.
rho = 3;
lWGrid = (lWHatMin:sigma_min/rho:lWHatMax)';
```

To guarantee that there exists a point in the wealth grid that matches $W(0)$, shift the logarithmic grid downwards by the smallest shift such that one of those values matches the logarithm of the initial portfolio value.

```
% Compute the shift for wealth grid.
lW0 = log(W0);
difference = lWGrid-lW0;
minDownShift = min(difference(difference>=0));
% Define the wealth nodes.
WGrid = exp(lWGrid - minDownShift);
nWealthNodes = size(WGrid,1); % Number of wealth nodes
```

Value Function and Optimal Strategy

The problem in this example is a multi-period optimization problem. The goal is to find the action at each rebalancing period that maximizes the probability that the wealth exceeds the target goal at the end of the investment period. To solve this problem, use the Bellman equation, which in this case is given by

$$V(W_i(t)) = \max_{\pi}\left[\sum_j V\big(W_j(t+1)\big)p\big(W_j(t+1)\,\big|\,W_i(t),\pi\big)\right].$$

The value function $V(W_i(t))$ represents the probability of exceeding the wealth goal G at time T, given that at time t, the wealth level is W_i. Simultaneously, $p\big(W_j(t+1)\,|\,W_i(t),\pi\big)$ represents the probability of moving to wealth node j at time $t+1$, given that at time t, you are at wealth node i and you chose portfolio π as the rebalancing action.

Compute Transition Probabilities

Since this example assumes that the wealth goal follows a geometric Brownian motion, the transition probabilities of being in wealth node W_j at time $t+1$, given that you are at wealth node W_i at time t, are time-homogeneous. Thus, the transition probability for a given portfolio choice π is defined as

$$p\big(W_j(t+1)\,|\,W_i(t),\pi\big) = \mathbb{P}\big(W_j \leq W(t+1) < W_{j+1}\,|\,W(t) = W_i,\pi\big) = \widetilde{p}\big(W_{j+1}\,|\,W_i,\pi\big) - \widetilde{p}\big(W_j\,|\,W_i,\pi\big),\ \ \forall\, t, \in \{0,\ldots,T\}$$

where

$$\widetilde{p}(W_k\,|\,W_i,\pi) = \Phi\left(\frac{1}{\sigma_\pi}\left(\ln\left(\frac{W_k}{W_i}\right) - \left(\mu_\pi - \frac{{\sigma_\pi}^2}{2}\right)\right)\right),$$

Φ is the cumulative distribution function of the standard normal random variable, and μ_π and σ_π are the return and risk of portfolio π. If $j = 1$,

$$p(W_1(t+1)\,|\,W_i(t),\pi) = \widetilde{p}(W_2\,|\,W_i,\pi),\ \ \forall\, t \in \{0,\ldots,T\},$$

and if $j = i_{\max}$,

$$p\big(W_{i_{\max}}(t+1)\,|\,W_i(t),\pi\big) = 1 - \widetilde{p}\big(W_{i_{\max}}\,|\,W_i,\pi\big),\ \ \forall\, t \in \{0,\ldots,T\}.$$

Note that this example computes transition probabilities differently from the transition probabilities as described by Das [1 on page 4-377], to keep a consistent mapping between $W(t)$ and a unique wealth node index.

```
% Compute the transition probabilities for different portfolios:
% pTransition(i,j,k) = p(Wj|Wi,mu_k)
pTransition = zeros(nWealthNodes,nWealthNodes,nPortfolios);
for k = 1:nPortfolios
    % Get the return and variance of portfolio k.
    mu = pret(k);
    sigma = prsk(k);

    % Compute p tilde before the cdf.
    pTilde = 1/sigma*(log(WGrid'./WGrid) - (mu-(sigma^2)/2));
    % Auxiliary matrices for edge cases (j = 1 and j = i_max)
    pTildeAuxUB = [pTilde(:,2:end) inf(nWealthNodes,1)];
    pTildeAuxLB = [-inf(nWealthNodes,1) pTilde(:,2:end)];

    % Compute the transition probabilities for portfolio k.
    pTransition(:,:,k) = normcdf(pTildeAuxUB)-normcdf(pTildeAuxLB);
end
```

Maximize Probability of Achieving Investor's Goal

Since the value function is the probability of achieving the wealth investment goal G at the final time step, you know that the value function at $t = T$ is given by

$$V(W_i(t)) = \begin{cases} 0 \text{ if } W_i(T) < G \\ 1 \text{ if } W_i(T) \geq G \end{cases}.$$

```
% Intialize the matrix that stores the value function at different
% wealth nodes and time periods.
V = zeros(nWealthNodes,T+1);
% Define the Value function at t = T.
V(:,end) = (WGrid >= G);
```

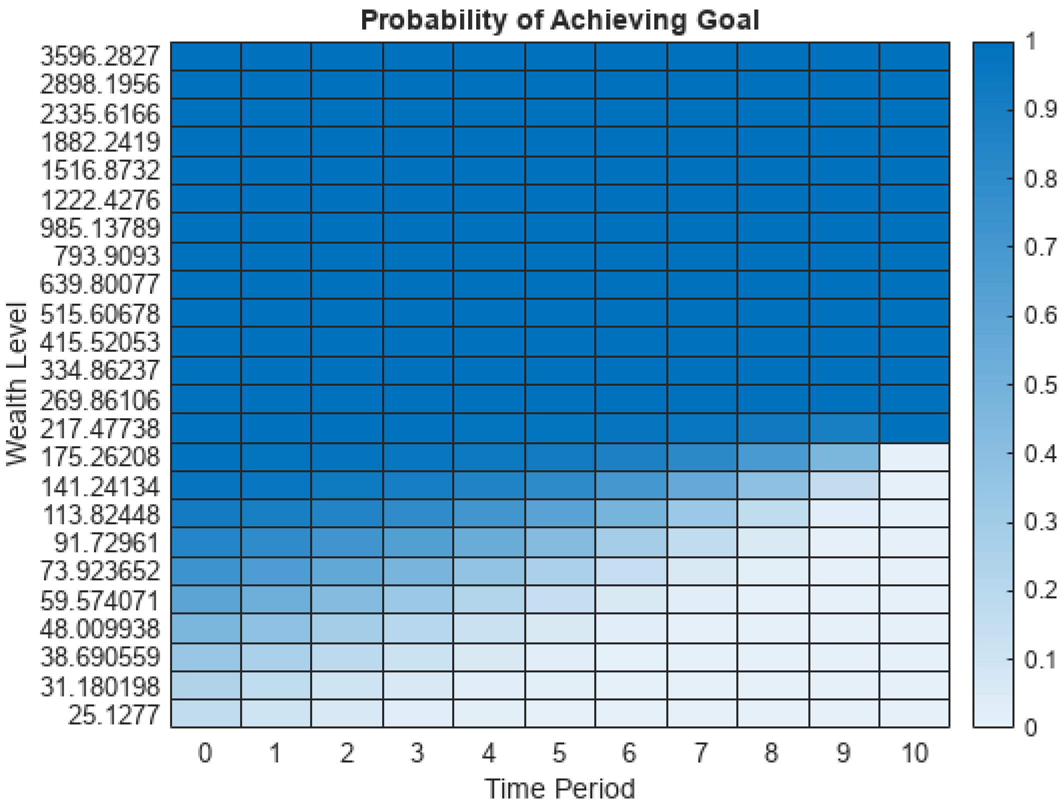
To compute the value of $V(W_i(t))$ for $t < T$, you need to go backwards in time. You start at $t = T - 1$ to determine the portfolio that achieves the maximum in the Bellman equation and $V(W_i(T-1))$. Then continue with $t = T - 2$, then $t = T - 3$, until you reach $t = 0$. The value of $V(W(0))$ is the optimal probability of attaining the wealth goal G from the initial wealth $W(0)$.

```
% Compute the optimal value function and portfolio at different
% wealth nodes and time periods.
portIdx = zeros(nWealthNodes,T);
for t = T:-1:1
    pTilde = zeros(nWealthNodes,nPortfolios);
    for k = 1:nPortfolios
        % sum_j ( V(Wj(t+1)) * p(Wj|Wi,mu_k) )
        pTilde(:,k) = pTransition(:,:,k)*V(:,t+1);
    end
    % Choose the portfolios that achieve the best probabilities.
    [V(:,t),portIdx(:,t)] = max(pTilde,[],2);
end
```

Note that `V(i,t)` is $V(W_i(t-1))$, which denotes the optimal probability of achieving the wealth goal from wealth node W_i at time period $t - 1$. On the other hand, `portIdx(i,t)` defines the optimal portfolio to be chosen at time period $t - 1$ if you are at wealth node W_i. The value of t is shifted down for `V(i,t)` and `portIdx(i,t)`, given that array indices start at `1` and t starts at `0`.

Use `heatmap` to plot the value function.

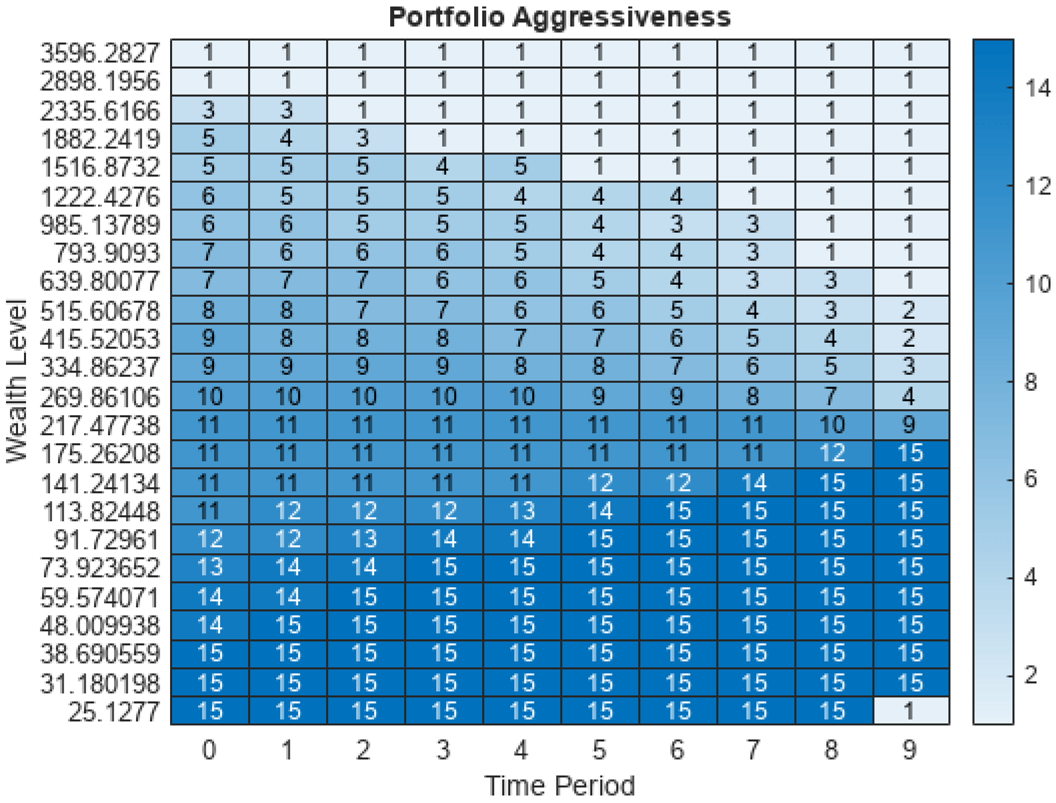
```
% Plot a heatmap of the value function.
figure;
heatmap(0:T,flip(WGrid(1:5:end)),flip(V(1:5:end,:)))
xlabel("Time Period")
ylabel("Wealth Level")
title("Probability of Achieving Goal")
```



The figure shows how the probability of achieving the goal becomes larger as the initial wealth grows. Also, this figure demonstrates that as time passes, the current wealth must be higher than in the previous periods to get the same probability of achieving the goal.

Use `heatmap` to plot the optimal portfolio.

```
% Plot a heatmap of optimal portfolio.
figure;
heatmap(0:T-1,flip(WGrid(1:5:end)),flip(portIdx(1:5:end,:)))
xlabel("Time Period")
ylabel("Wealth Level")
title("Portfolio Aggressiveness")
```



	0	1	2	3	4	5	6	7	8	9
3596.2827	1	1	1	1	1	1	1	1	1	1
2898.1956	1	1	1	1	1	1	1	1	1	1
2335.6166	3	3	1	1	1	1	1	1	1	1
1882.2419	5	4	3	1	1	1	1	1	1	1
1516.8732	5	5	5	4	5	1	1	1	1	1
1222.4276	6	5	5	5	4	4	4	1	1	1
985.13789	6	6	5	5	5	4	3	3	1	1
793.9093	7	6	6	6	5	4	4	3	1	1
639.80077	7	7	7	6	6	5	4	3	3	1
515.60678	8	8	7	7	6	6	5	4	3	2
415.52053	9	8	8	8	7	7	6	5	4	2
334.86237	9	9	9	9	8	8	7	6	5	3
269.86106	10	10	10	10	10	9	9	8	7	4
217.47738	11	11	11	11	11	11	11	11	10	9
175.26208	11	11	11	11	11	11	11	11	12	15
141.24134	11	11	11	11	11	12	12	14	15	15
113.82448	11	12	12	12	13	14	15	15	15	15
91.72961	12	12	13	14	14	15	15	15	15	15
73.923652	13	14	14	15	15	15	15	15	15	15
59.574071	14	14	15	15	15	15	15	15	15	15
48.009938	14	15	15	15	15	15	15	15	15	15
38.690559	15	15	15	15	15	15	15	15	15	15
31.180198	15	15	15	15	15	15	15	15	15	15
25.1277	15	15	15	15	15	15	15	15	15	1

This figure shows that the higher the wealth, the lower the number of the optimal portfolio. Since the number of the portfolio is directly related to the aggressiveness of the strategy, you can see that the wealthier the portfolio, the less risk is needed to achieve the final wealth goal. When the portfolio has less money, then the optimal portfolio is the one with a larger expected return, even at the cost of increasing the volatility. Also, note that the closer the time period is to the end of the investment horizon, the choice of the optimal portfolio becomes polarized, where the portfolio is either extremely aggressive or extremely conservative.

Implement Optimal Strategy

The following describes the workflow to implement the optimal strategy:

1. At $t = 0$, choose the optimal strategy shown in `portIdx(i,j)` such that i is the index asociated with the wealth node that satisfies $W_i \leq W(0) < W_{i+1}$ and $j = t + 1$. Also, let $A^*(0) =$ `portIdx(i,j)`.
2. Rebalance the portfolio to match the weights in `pwgt(:,`$A^*(0)$`)`. Then, define $\pi^*(0) =$`pwgt(:,` $A^*(0)$`)`.
3. Either using historical data for time period $t = 0$ or simulating one step of a Brownian motion, obtain $W(1)$, given that you started at $W(0)$, and chose portfolio $\pi^*(0)$.
4. Repeat steps 1 through 3 for $t = 1, \ldots, T - 1$.

```
% Simulate one path following the optimal strategy.
rng('default')
[Z,WPath,portPath,VPath] = DynamicPortSim(W0,WGrid,pret,prsk, ...
    portIdx,V);
```

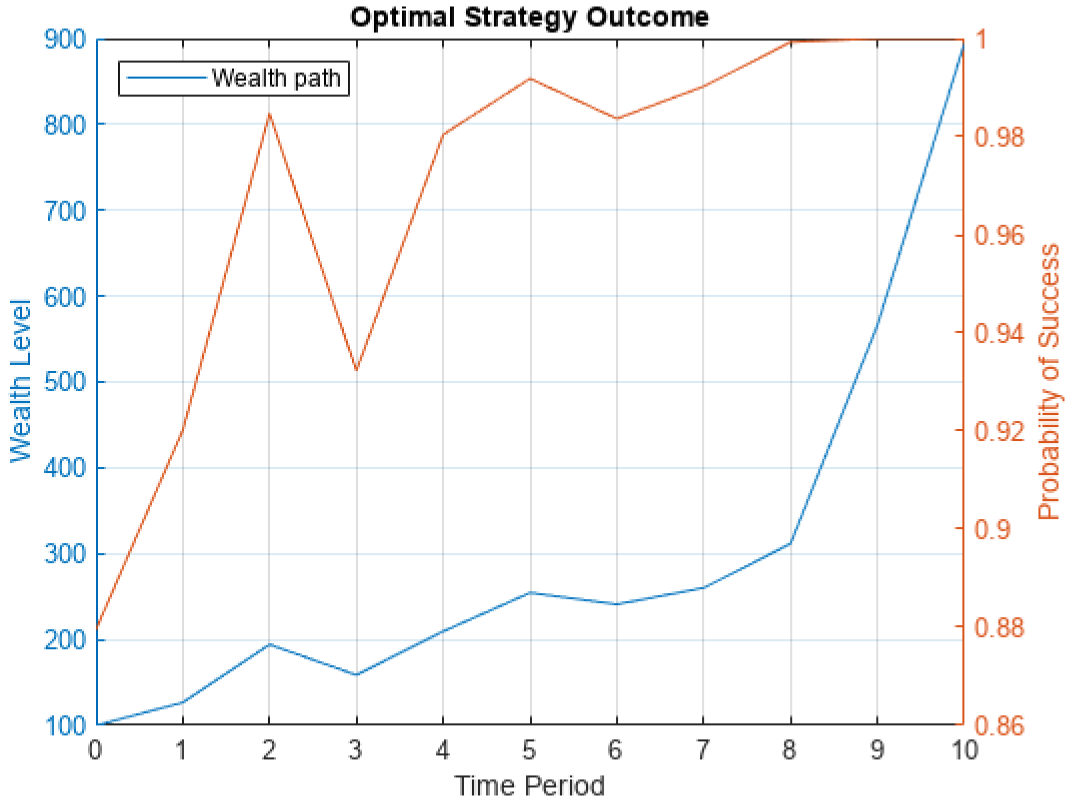
Present a table with the results of the simulation.

```
simulation = table((0:T)',Z,WPath,portPath,VPath,'VariableNames', ...
    {'t','Z','Wealth','OptPort','ProbOfSuccess'})
```

```
simulation=11×5 table
    t        Z         Wealth    OptPort    ProbOfSuccess
    __    ________    ______    _______    _____________

     0     0.53767       100       12          0.87953
     1      1.8339    126.69       11          0.92018
     2     -2.2588    194.27       11          0.98474
     3     0.86217    158.68       11           0.9323
     4     0.31877    209.52       11          0.98044
     5     -1.3077    254.46       10          0.99184
     6    -0.43359    241.04       10          0.98366
     7     0.34262    260.14        8          0.99024
     8      3.5784    311.62        5          0.99927
     9      2.7694    566.28        1                1
    10         NaN     894.5      NaN                1
```

Note that as the probability of attaining the wealth goal (the value function) increases, the aggressiveness (number) of the optimal portfolio decreases. Now, you can plot the path of the wealth against the probability of achieving the wealth goal at the end of the investment period.

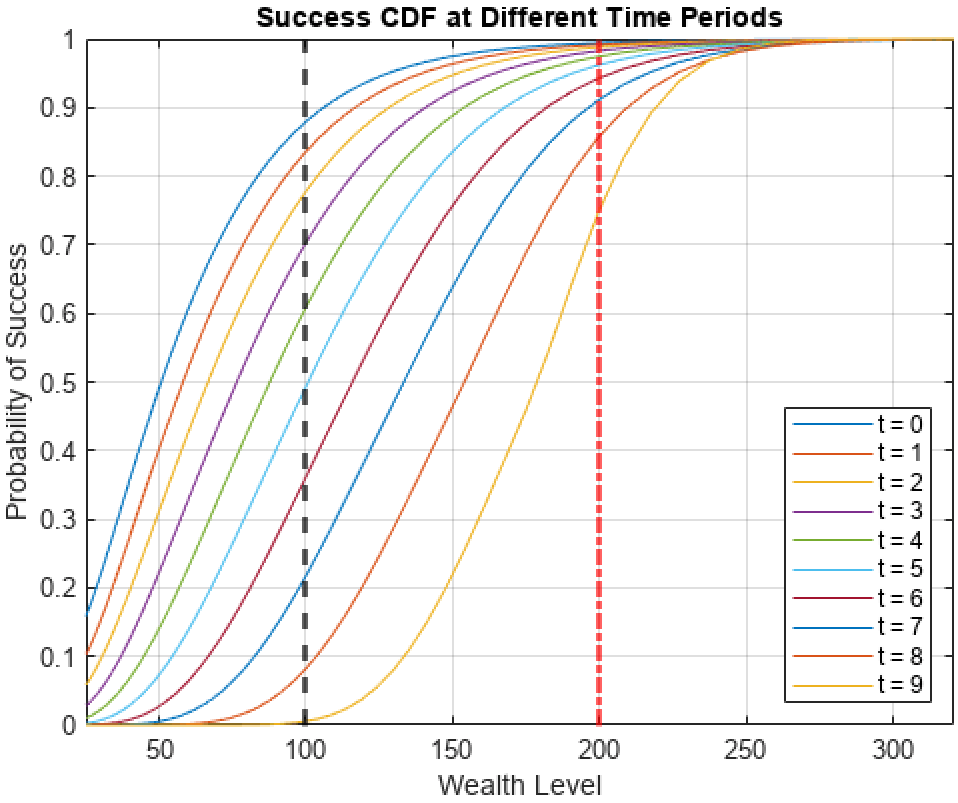
```
figure;

% Plot the wealth path.
yyaxis left
plot(0:T,WPath)
hold on
yline(G,'--k');
ylabel("Wealth Level");

% Plot the value function path.
yyaxis right
plot(0:T,VPath)
ylabel("Probability of Success");

% Specify the title, labels, and legends for the plot.
title("Optimal Strategy Outcome")
xlabel("Time Period");
legend("Wealth path","Wealth goal",Location="nw");
grid on
hold off
```



At $t = 4$, the wealth goal has been achieved. However, because of the uncertainty in the problem, the probability of success is not yet 1. Furthermore, trying different paths (random seeds) shows how even a monotonic increase in the wealth does not translate to a monotonic increase in the probability of success. This behavior occurs because the wealth increase at later time periods should be higher than at earlier time periods to have the same increase in the probability of success.

Plot the cumulative distribution function (CDF) of attaining the goal wealth at different time periods.

```
% Plot the cumulative distribution function (CDF) of success at
% different time periods.
figure
nWN = 60;
plot(WGrid(1:nWN),V(1:nWN,1:end-1))
xline(W0,'--k','LineWidth',2); % Initial wealth
xline(G,'-.r','LineWidth',2); % Goal wealth
xlim(WGrid([1,nWN]))

% Specify the title, labels, and legends.
title('Success CDF at Different Time Periods')
xlabel('Wealth Level')
ylabel('Probability of Success')
legend("t = "+num2str((0:T-1)'),Location="se")
grid on
```



This figure shows how as the time period approaches the investment horizon, the CDF shifts to the right and becomes steeper. This shift demonstrates that changes in the wealth level have a higher impact toward the end of the investment horizon. For example, an increase of 100 to 110 wealth points at *t* = 0 only increases the probability of success by 3%, while the same wealth increase at *t* = 4 increases the probability of success by 7%.

Local Functions

```
function [Z,WPath,portPath,VPath] = DynamicPortSim(W0,WGrid,...
    pret,prsk,portIdx,V)
% Define time periods.
T = size(V,2)-1;

% Define standard normal random variables for the Brownian motion.
Z = randn(T,1);
Z = [Z;nan]; % No randomness at the final time period

% Find the wealth node index of the initial wealth.
tol = 1e-8; % Account for numerical inaccuracies
difference = WGrid-W0;
% Assign to wealth node j such that W_j <= W(0) < W_{j+1}
WNodeIdx = find(difference<=tol,1,'last');

% Run the simulation.
WPath = [W0; zeros(T,1)];
portPath = [portIdx(WNodeIdx,1); zeros(T,1)];
VPath = [V(WNodeIdx,1); zeros(T,1)];
```

```
for t = 1:T
    % Get the return and variance of optimal portfolio at time t.
    mu = pret(portIdx(WNodeIdx,t));
    sigma = prsk(portIdx(WNodeIdx,t));

    % Compute the wealth at time t+1 following the Browninan motion.
    WPath(t+1) = WPath(t)*exp(mu-(sigma^2)/2+sigma*Z(t));

    % Compute the wealth node index corresponding to the current wealth.
    difference = WGrid-WPath(t+1);
    % Assign to wealth node j such that W_j <= W(t+1) < W_{j+1}
    WNodeIdx = find(difference <= tol,1,'last');

    % Store the optimum strategy and value function.
    if t<T
        portPath(t+1) = portIdx(WNodeIdx,t+1);
    else
        portPath(t+1) = nan;
    end
    VPath(t+1) = V(WNodeIdx,t+1);
end
end
```

References

[1] Das, Sanjiv R., Daniel Ostrov, Anand Radhakrishnan, and Deep Srivastav. “Dynamic Portfolio Allocation in Goals-Based Wealth Management.” *Computational Management Science* 17, no. 4 (December 2020): 613–40. Available at: https://doi.org/10.1007/s10287-019-00351-7.

See Also

`Portfolio` | `estimatePortSharpeRatio` | `estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierByRisk`

Related Examples

- “Single Period Goal-Based Wealth Management” on page 4-361

Compare Performance of Covariance Denoising with Factor Modeling Using Backtesting

This example uses backtesting to compare the performance of two investment strategies that use factor information to compute the portfolio weights. The first investment strategy uses `covarianceDenoising` to estimate both the covariance matrix and the number of factors to use in the second investment strategy. The second investment strategy uses a principal component analysis (PCA) factor model to estimate the covariance matrix with the number of factors obtained with `covarianceDenoising`. The PCA factor model follows the process in “Portfolio Optimization Using Factor Models” on page 4-224.

Load Data

Load a simulated data set that includes asset returns for a total n = 100 assets and 2000 daily observations.

```
load('asset_return_100_simulated.mat');
[numObservations,numAssets] = size(stockReturns)
```

```
numObservations = 2000
```

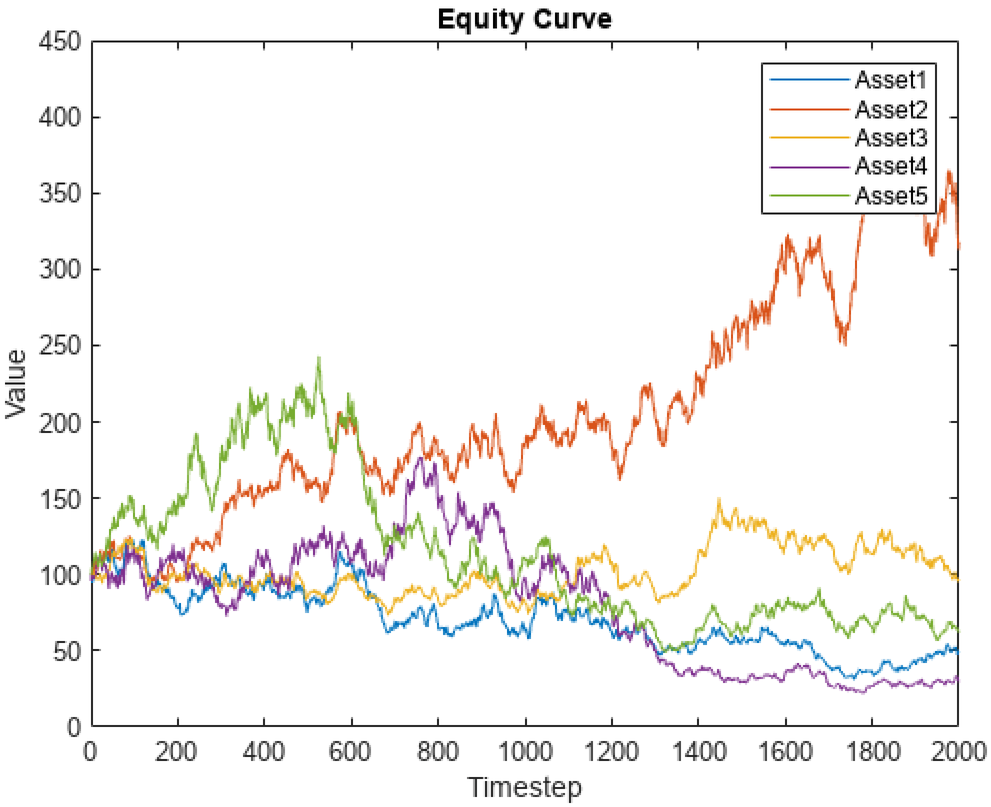
```
numAssets = 100
```

Create a timetable of asset prices from the asset returns.

```
% Convert the returns to prices
pricesT = ret2tick(stockReturns,'StartPrice',100);

% Create timetable
rowTimes = datetime("today"):datetime("today")+numObservations;
pricesTT = table2timetable(pricesT,'RowTimes',rowTimes);
```

Visualize the equity curve for each stock. For this example, plot the first five stocks.

```
figure;
plot(0:2000,pricesTT{:,1:5})
xlabel('Timestep');
ylabel('Value');
title('Equity Curve');
legend(pricesTT.Properties.VariableNames(1:5));
```



Optimize Asset Allocation Using Covariance Denoising

Covariance denoising is a technique that you can use to reduce the noise and enhance the signal in a covariance matrix. First, the eigenvalues that are associated with noise are separated from the eigenvalues associated with signal. Then, the eigenvalues associated with noise are shrunk towards a target value. This technique helps improve the stability of the covariance matrix over time as well as its condition number.

The function `covarianceDenoising` computes the denoised estimate of the covariance matrix and returns as a second output the number of eigenvalues identified with signal. You use this number in the Optimize the Asset Allocation Using Factor Modeling on page 4-380 section to determine the number of factors that the factor model allocation uses.

This example uses the first 42 days (approximately 2 months) of the data set to select the initial portfolio allocations.

```
% Warm-up period
warmupPeriod = 42;

% No current weights (100% cash position)
w0 = zeros(1,numAssets);

% Warm-up partition of prices timetable
warmupTT = pricesTT(1:warmupPeriod,:);
```

Compute the maximum return portfolio subject to a target risk of `0.008` using the denoised covariance estimate.

```
% Compute weights with denoised strategy
wDenoised_initial = denoising(w0,warmupTT);
```

Check for asset allocations that are over 5% to identify assets with large investment weights.

```
percentage = 0.05;
AssetName = pricesTT.Properties.VariableNames(...
    wDenoised_initial>=percentage)';
Weight = wDenoised_initial(wDenoised_initial>=percentage);
T1 = table(AssetName,Weight)
```

```
T1=5×2 table
     AssetName      Weight 
    ___________    ________

    {'Asset6' }    0.066014
    {'Asset47'}     0.10991
    {'Asset50'}     0.24654
    {'Asset75'}     0.11752
    {'Asset94'}     0.31708
```

Optimize Asset Allocation Using Factor Modeling

For factor modeling, you can use statistical factors extracted from the asset return series. In this example, PCA is used to extract these factors [1 on page 4-384]. You can then use this factor model to solve the portfolio optimization problem.

With a factor model, n asset returns can be expressed as a linear combination of k factor returns, $r_a = \mu_a + F\, r_f + \varepsilon_a$, where $k \ll p$. In the mean-variance framework, portfolio risk is

$$\mathrm{Var}\left(R_p\right) = \mathrm{Var}\left({r_a}^T w_a\right) = \mathrm{Var}\left(\left(\mu_a + F\, r_f + \varepsilon_a\right)^T w_a\right) = w_a^T\left(F\Sigma_f F^T + D\right) w_a,$$

where:

- R_p is the portfolio return (a scalar).
- r_a is the asset returns.
- μ_a is the mean of asset returns.
- F is the factor loading, with dimension $n \times k$.
- r_f is the factor return.
- ε_a is the idiosyncratic return related to each asset.
- w_a is the asset weight.
- Σ_f is the covariance of factor returns.
- D is the variance of idiosyncratic returns.

The parameters r_a, w_a, μ_a and ε_a are $n \times 1$ column vectors, $r_f$and w_f are $k \times 1$ column vectors, and Σ_k and D are a $k \times k$ and a $n \times n$ matrices, respectively.

Therefore, the mean-variance optimization problem is formulated as

$$\begin{aligned}
&\max_{w_a} \ \mu_a^T w_a \\
&s.t. \ w_a^T\left(F\Sigma_f F^T + D\right)w_a \le \tau, \\
&\qquad \sum_{a \in A} w_a = 1, \\
&\qquad 0 \le w_a \le 1 .
\end{aligned}$$

In the dimensional space formed by n asset returns, PCA finds the k directions that capture the most important variations in the returns. Usually, k is less than n. Therefore, by using PCA, you can decompose the n asset returns into k directions that are interpreted as factor loadings. The scores from the decomposition are interpreted as the factor returns. For more information, see `pca` (Statistics and Machine Learning Toolbox™). In this example, the factor model uses k =`nFactors`, where `covarianceDenoising` determines the `numFactors`.

Compute the maximum return portfolio subject to a target risk of 0.008 using the factor model covariance estimate. For details on how to obtain the weights allocation using factor modeling, see “Portfolio Optimization Using Factor Models” on page 4-224.

```
% Compute weights with denoised strategy
userData.numFactors = [];
[wFactorModel_initial,userData] = factorModeling(w0,warmupTT, ...
    userData);
```

Check for asset allocations that are over 5% to show assets with large investment weights.

```
percentage = 0.05;
AssetName = pricesTT.Properties.VariableNames( ...
    wFactorModel_initial>=percentage)';
Weight = wFactorModel_initial(wFactorModel_initial>=percentage);
T2 = table(AssetName,Weight)
```

```
T2=6×2 table
     AssetName       Weight 
    ___________    ________

    {'Asset6' }    0.075366
    {'Asset35'}    0.069395
    {'Asset47'}     0.10676
    {'Asset50'}     0.21628
    {'Asset75'}     0.12423
    {'Asset94'}      0.3068
```

The assets with large investment weights are almost the same for both investment strategies. `Asset35` is the only asset that appears in one table, namely in the factor model strategy, and not the other. Even the weights of the assets are similar.

Backtesting

Use `backtestStrategy` to create strategy objects for the two investment strategies. Compare the denoising strategy (`strat1`) against the factor model strategy (`strat2`) using backtesting.

```
% Rebalance approximately every month
rebalFreq = 21;
```

```
% Set the rolling lookback window to be at least 2 months and at
% most 6 months
lookback  = [42 126];

% Use a fixed transaction cost (buy and sell costs are both 0.5%
% of amount traded)
transactionsFixed = 0.005;

% Strategies
strat1 = backtestStrategy('Factor Modeling', @factorModeling, ...
    UserData=userData, ...
    RebalanceFrequency=rebalFreq, ...
    LookbackWindow=lookback, ...
    TransactionCosts=transactionsFixed, ...
    InitialWeights=wFactorModel_initial);

strat2 = backtestStrategy('Denoising', @denoising, ...
    RebalanceFrequency=rebalFreq, ...
    LookbackWindow=lookback, ...
    TransactionCosts=transactionsFixed, ...
    InitialWeights=wDenoised_initial);

% Aggregate the strategy objects into an array
strategies = [strat1, strat2];
```

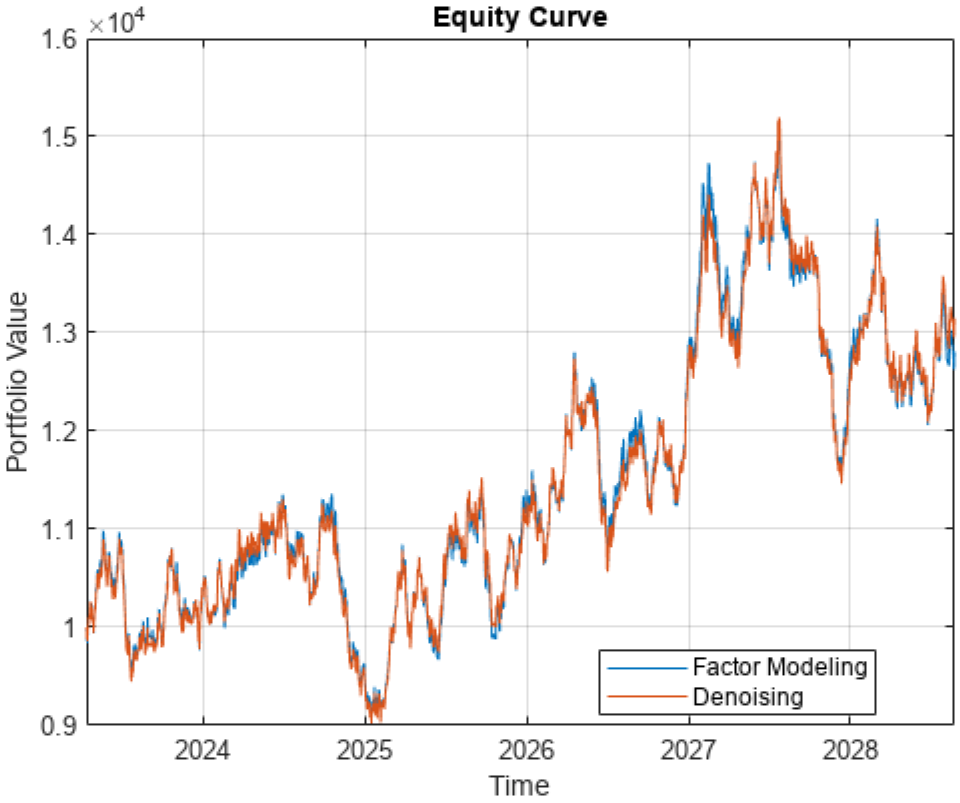
Create a `backtestEngine` object for the strategies, run the backtest using `runBacktest`, and generate a report using `summary`.

```
% Create the backtesting engine object
backtester = backtestEngine(strategies);

% Run backtest
backtester = runBacktest(backtester,pricesTT,'Start',warmupPeriod);

% Generate summary table of strategies performance
summary(backtester)
```

```
ans=9×2 table
                       Factor_Modeling    Denoising 
                       _______________    __________

    TotalReturn              0.28079         0.31501
    SharpeRatio             0.018583        0.020462
    Volatility             0.0089592       0.0086696
    AverageTurnover         0.014646        0.013843
    MaxTurnover              0.56184         0.58224
    AverageReturn         0.00016644      0.00017736
    MaxDrawdown              0.23384         0.24536
    AverageBuyCost           0.84446         0.79536
    AverageSellCost          0.84446         0.79536
```

Use `equityCurve` to plot the equity curve to compare the performance of both strategies.

```
equityCurve(backtester)
```



The performance of both strategies is similar, although not identical. This similarity is because the factor model strategy uses the number of factors identified by `covarianceDenoising` to select the number of principal components. Factor model strategies are usually not implemented this way, but rather the number of factors is a fixed parameter that is chosen *a priori*.

In this example, the number of factors that is most frequently identified is `1`.

```
% Count the number of different factors
categoricalNumFactors = ...
    categorical(backtester.Strategies(1).UserData.numFactors);
[N,uniqueFactors] = histcounts(categoricalNumFactors);
factorFrequency = table(uniqueFactors',N', ...
    'VariableNames',{'NumFactors','Frequency'})
```

```
factorFrequency=3×2 table
    NumFactors    Frequency
    __________    _________

      {'1' }         81    
      {'2' }         12    
      {'17'}          1    
```

Therefore, you can run the factor modeling strategy using `1` as the number of factors instead of using `covarianceDenoising` to identify the number of factors at each rebalancing period.

Reference

1 Meucci, Attilio. “Modeling the Market.” In *Risk and Asset Allocation*, by Attilio Meucci, 101–66. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.

Local Functions

```
function [new_weights,userData] =...
    factorModeling(~,pricesTT,userData)
% Compute minimum variance portfolio using traditional covariance estimate.

% Compute returns from prices timetable.
assetReturns = tick2ret(pricesTT);

% Compute the number of factors identified using covariance denoising.
[~,numFactors] = covarianceDenoising(assetReturns.Variables);
userData.numFactors = [userData.numFactors; numFactors];

% Compute the covariance using the factors model
%   SigmaFactorModel = F*Sigma_f*F' + D
%   r_a = mu_a + F*r_f + epsilon_a
[factorLoading,factorRetn,~,~,~,factorMean] = ...
    pca(assetReturns.Variables,'NumComponents',numFactors);
covFactor = cov(factorRetn);
retnHat = factorRetn*factorLoading' + factorMean;
unexplainedRetn = assetReturns.Variables - retnHat;
unexplainedCovar = diag(cov(unexplainedRetn));
D = diag(unexplainedCovar);

% Define the mean and covariance of the returns.
mu = mean(assetReturns.Variables);
Sigma = factorLoading*covFactor*factorLoading' + D;

% Create the portfolio problem.
p = Portfolio(AssetMean=mu,AssetCovar=Sigma);
% Specify long-only, fully-invested contraints
p = setDefaultConstraints(p);

% Compute the maximum return portfolio subject to the target risk.
targetRisk = 0.008;
new_weights = estimateFrontierByRisk(p,targetRisk);
end

function new_weights = denoising(~, pricesTT)
% Compute minimum variance portfolio using covariance denoising.

% Compute the returns from the prices timetable.
assetReturns = tick2ret(pricesTT);
mu = mean(assetReturns.Variables);
Sigma = covarianceDenoising(assetReturns.Variables);

% Create the portfolio problem.
p = Portfolio(AssetMean=mu,AssetCovar=Sigma);
% Long-only fully invested contraints
p = setDefaultConstraints(p);
```

```
% Compute maximum return portfolio subject to the target risk.
targetRisk = 0.008;
new_weights = estimateFrontierByRisk(p,targetRisk);
end
```

See Also

`Portfolio` | `covarianceShrinkage` | `covarianceDenoising`

Related Examples

- “Portfolio Optimization Against a Benchmark” on page 4-195

More About

- “Comparison of Methods for Covariance Estimation” on page 4-134

Deep Reinforcement Learning for Optimal Trade Execution

This example shows how to use the Reinforcement Learning Toolbox™ and Deep Learning Toolbox™ to design agents for optimal trade execution.

Introduction

Optimal trade execution seeks to minimize trading costs when selling or buying a set number of stock shares over a certain time horizon within the trading day. Optimization is necessary because executing trades too quickly results in suboptimal prices due to market impact, while executing trades too slowly results in exposure to the risk of adverse price changes or the inability to execute all trades within the time horizon. This optimization affects nearly all trading strategies and portfolio management practices, since minimizing trading costs is directly linked to profitability that is related to buying or selling decisions. Also, brokers compete to provide better trade order execution, and many jurisdictions legally require brokers to provide the best execution for their clients. Early studies by Bertsimas and Lo [1 on page 4-419] and Almgren and Chriss [2 on page 4-419] derived analytical solutions for the optimal trade execution problem by assuming a model for the underlying prices. Later, Nevmyvaka et al. [3 on page 4-419] demonstrated that the use of reinforcement learning (RL) for optimal execution did not require making assumptions about the market micro-structure. More recent studies use deep reinforcement learning,which combines reinforcement learning (RL) with deep learning, for optimal trade execution (Ning *et al*. [4 on page 4-419], Lin and Beling [5 on page 4-419,6 on page 4-419]). This approach overcomes the limitations of Q-learning as noted by Nevmyvaka et al. [3 on page 4-419].

This example uses deep reinforcement learning to design and train two deep Q-network (DQN) agents for optimal trade execution. One DQN agent is for sell trades and the other agent is for buy trades.

Data

This example uses the intraday trading data on Intel Corporation stock (INTC) provided by LOBSTER [7 on page 4-420] and included with Financial Toolbox™. While the reinforcement learning for optimal execution literature typically uses trading data over one year or more to train and test RL agents [3 on page 4-419,4 on page 4-419,5 on page 4-419,6 on page 4-419], in this proof-of-concept example you use a shorter data set, which you can expand for further study. The intraday trading data on the Intel stock (`INTC_2012-06-21_34200000_57600000_orderbook_5.csv`) contains limit order book data over 6.5 hours, including bid and ask prices as well as corresponding shares for 5 levels. This data is sampled at 5 second intervals to create 390 one minute trading horizons, such that each trading horizon contains 12 steps at five second intervals. Out of these 390 horizons, you use 293 horizons (about 75 percent) to train the agents and the remaining 97 horizons (about 25 percent) to test the agents.

Baseline Trade Execution Algorithm

An agent's performance must be compared with a baseline trade execution algorithm. For this example, the baseline trade execution algorithm is the time-weighted average price (TWAP) policy. Under this policy, the total trading shares are divided equally over the trading horizon, so that the same number of shares are traded at each time step. In contrast, the original paper by Nevmyvaka *et al*. [3 on page 4-419] used a "submit and leave" policy as the baseline, where a limit order is placed at a fixed price for the entire trading shares for the entire horizon, and any remaining unfilled shares are traded with a market order at the last step. However, the TWAP baseline was adopted in later studies [4 on page 4-419,5 on page 4-419,6 on page 4-419], since it has become more popular in recent years with increasing use of algorithmic trading, and it is also a highly effective strategy that is thought to be optimal when the price movement follows a Brownian motion [4 on page 4-419,5 on page 4-419,6 on page 4-419].

Agents

Different types of agents are used in the reinforcement learning for optimal execution studies, including Q-learning [3 on page 4-419], DQN [4 on page 4-419,5 on page 4-419], and PPO [6 on page 4-419] agents. In this example, you design and train DQN agents using the Reinforcement Learning Designer (Reinforcement Learning Toolbox) after creating separate training environments for sell trades and buy trades. These DQN agents only have a critic without an actor, and they use the default setting of `UseDoubleDQN` set to `true`, while the `DiscountFactor` is set to `1` due to the short trading horizon [4 on page 4-419,5 on page 4-419,6 on page 4-419]. In addition, the critic network of these agents include a long short-term memory (LSTM) network because the trading data is sequential in time, and LSTM networks are effective in designing agents for optimal execution [6 on page 4-419]. For more information on setting options for DQN agents, see `rlDQNAgentOptions` (Reinforcement Learning Toolbox).

Observation Space

The observation space defines which information (or feature) to provide to the agents at each step. The agents use this observation to decide the next action, based on the policy learned and reinforced by the rewards received after taking the previous actions. Designing a good observation space for the agents is an area of active research in optimal execution studies. You can experiment with different observation spaces, using variables such as the bid-ask spread, the bid or ask volume, price trends, or even the entire limit order book prices and shares [6 on page 4-419]. This example uses the following observation variables:

- Remaining shares to be traded by the agent in the current trading horizon
- Remaining time intervals (steps) left in the current trading horizon
- Cumulative implementation shortfall (IS) of agent until the previous step
- Current limit order book price divergence from arrival price

The first two observation variables (remaining shares to be traded and remaining time in horizon) are called "private" variables by Nevmyvaka *et al*. [3 on page 4-419], as they only apply to the specific situation of the trading agents, and they do not apply to other market participants. As the agent trained by Nevmyvaka *et al*. [3 on page 4-419] using only "private" variables was able to outperform the "submit and leave" baseline policy, the "private" variables are also used in the subsequent studies [4 on page 4-419,5 on page 4-419,6 on page 4-419].

The third variable, cumulative IS, is a measure of trading cost that the agents seek to minimize. As the name "shortfall" indicates, a lower implementation shortfall implies better trade execution. Technically, Implementation Shortfall should include the opportunity cost of failing to execute all shares within the trading horizon. In this example, as with many of the previously mentioned studies, the Implementation Shortfall is computed under the ending inventory constraint:

- For a sell trade, the constraint is to have zero ending inventory, assuming successful liquidation of all shares.
- For a buy trade, the constraint is to have full ending inventory, assuming successful purchase of all shares.

Under this ending inventory constraint, Implementation Shortfall is computed as follows:

- For a sell trade:
  $\text{Implementation Shortfall} = \text{Arrival Price} \times \text{Traded Volume} - \text{Executed Price} \times \text{Traded Volume}$
- For a buy trade:
  $\text{Implementation Shortfall} = \text{Executed Price} \times \text{Traded Volume} - \text{Arrival Price} \times \text{Traded Volume}$

The arrival price is the first bid or ask price observed at the beginning of the trade horizon. If Implementation Shortfall is positive, it implies that the average executed price was worse than the arrival price, while a negative Implementation Shortfall implies that the average executed price was better than the arrival price. The cumulative Implementation Shortfall from the beginning of the trade horizon until the previous step reflects the trading cost incurred by the agent in the past, and it serves as the third observation variable.

The last observation variable is the current price divergence from arrival price. This variable reflects the current market condition and it is measured as the difference between the arrival price and the average price of the first two levels of the current limit order book (LOB).

- For a sell trade:
  $\text{Price Divergence} = \text{Average Bid Prices of First 2 Levels of Current LOB} - \text{Arrival Price}$
- For a buy trade:
  $\text{Price Divergence} = \text{Arrival Price} - \text{Average Ask Prices of First 2 Levels of Current LOB}$

A positive price divergence implies more favorable current trading conditions than the arrival price, and a negative divergence less favorable conditions.

Action Space

The action space is defined as the possible number of shares traded at each step. For a TWAP policy, the only possible action at each step is to trade a constant number of shares computed as the total trading shares divided by the number of steps in the horizon. Meanwhile, the agents in this example choose from 39 possible actions that are mostly evenly spaced, ranging from trading 0 shares to trading twice the number of shares as the corresponding TWAP trade.

For both TWAP trades and agent trades, you enforce ending inventory constraints so that if the target ending inventory is not met by the last step of the horizon, the action for the last step is to:

- Sell any remaining shares in the inventory for a Sell trade.
- Buy any missing shares in the inventory for a Buy trade.

Also, you place limits on the agent actions to take advantage of price divergence and to prevent selling more than the available shares in the inventory or buying more than the target number of shares.

Reward

The reward is another important aspect of agent design, as agents learn their policies at each step using the rewards received after taking the previous actions. Research on reward design shows wide variations in optimal execution, and you can experiment with different rewards. Since the agent performance is measured against the baseline using Implementation Shortfall, many studies use Implementation Shortfalls directly in the rewards, while others use related quantities. Some studies use the proceeds from the trade as the reward [3 on page 4-419], while others also impose a penalty for trading many shares too quickly [4 on page 4-419]. Another study uses an elaborately shaped reward algorithm that is designed to use the TWAP policy most of the time, while encouraging the agent to deviate from the TWAP policy only when it is advantageous to do so [5 on page 4-419]. If there is enough data, the reward may be given sparsely, only at the end of each horizon rather than at each step [6 on page 4-419]. In this example, the data size is small, so you give the reward at each step and design it so that the reward at each step approximates the expected reward at the end of the horizon.

Specifically, you use a reward that compares the Implementation Shortfall of the TWAP baseline against that of the agent at each step, while also taking into account the penalty for the approximate Implementation Shortfall of the remaining unexecuted shares based on the current limit order book. The reward is also scaled based on the current time relative to the end of the horizon.

$$\text{Reward}(t) = \left[\left(\text{IS}_{\text{TWAP}}(0,t) + \text{IS}_{\text{TWAP, Penalty}}(t,T)\right) - \left(\text{IS}_{\text{Agent}}(0,t) + \text{IS}_{\text{Agent, Penalty}}(t,T)\right)\right] \times \frac{t}{T}$$

$\text{Reward}(t)$ is the reward at time step t

$\text{IS}_{\text{TWAP}}(0,t)$ is the Implementation Shortfall of TWAP from the beginning of the horizon until time step t

$\text{IS}_{\text{TWAP, Penalty}}(t,T)$ is the approximate Implementation Shortfall of TWAP from time step t to the end of the horizon at time step T

$\text{IS}_{\text{Agent}}(0,t)$ is the Implementation Shortfall of the agent from the beginning of horizon until time step t

$\text{IS}_{\text{Agent, Penalty}}(t,T)$ is the approximate Implementation Shortfall of the agent from time step t to the end of the horizon at time step T

Since the agent performance is ultimately measured against the baseline using Implementation Shortfall at the end of the horizon at time step T, and the approximate reward becomes more accurate as t approaches T, $\text{Reward}(t)$ has a scaling factor $\frac{t}{T}$ so that its magnitude increases as t approaches T. At time step T, the reward becomes $\text{Reward}(T)$.

$$\text{Reward}(T) = \text{IS}_{\text{TWAP}}(0,T) - \text{IS}_{\text{Agent}}(0,T)$$

Performance Analysis

You measure agent performance against the TWAP baseline using Implementation Shortfall (IS), by computing the following for all horizons:

- IS total outperformance of agent over TWAP — Sum of $\text{IS}_{\text{TWAP}}(0,T) - \text{IS}_{\text{Agent}}(0,T)$
- IS total-gain-to-loss ratio (TGLR) — Ratio of total positive over total negative $\text{IS}_{\text{TWAP}}(0,T) - \text{IS}_{\text{Agent}}(0,T)$ magnitudes
- IS gain-to-loss ratio (GLR) — Ratio of mean positive over mean negative $\text{IS}_{\text{TWAP}}(0,T) - \text{IS}_{\text{Agent}}(0,T)$ magnitudes

Workflow Organization

The workflow in this example is:

- **Load and Prepare Data** on page 4-390
- - LOBSTER Message Data — Contains time stamps

- - LOBSTER Limit Order Book Data — Contains bid and ask prices and corresponding shares
- **Optimal Execution for Sell Trades** on page 4-391
- - TWAP Baseline
- - Train the DQN Agent
- - Test the DQN Agent
- - Summary of Selling Results
- **Optimal Execution for Buy Trades** on page 4-415
- **Local Functions** on page 4-420
- - `executeTWAPTrades` — Executes TWAP Baseline trades and computes IS
- - `RL_OptimalExecution_LOB_ResetFcn` — Reset function for training and testing environments
- - `RL_OptimalExecution_LOB_StepFcn` — Step function for training and testing environments
- - `tradeLOB` — Executes trades on current limit order book
- - `endingInventoryIS` — Computes IS with ending inventory constraints
- - `ISTGLR` — Computes total-gain-to-loss ratio and gain-to-loss ratio for IS

Load and Prepare Message Data

Extract the LOBSTER data files from the ZIP file included with Financial Toolbox™. The expanded files include two CSV files of data and the text file `LOBSTER_SampleFiles_ReadMe.txt`.

```
unzip("LOBSTER_SampleFile_INTC_2012-06-21_5.zip");
```

The limit order book file contains the bid and ask prices and corresponding shares for five levels, while the message file contains the description of the limit order book data including time stamps.

```
LOBFileName = 'INTC_2012-06-21_34200000_57600000_orderbook_5.csv'; % Limit Order Book Data file
MSGFileName = 'INTC_2012-06-21_34200000_57600000_message_5.csv'; % Message file (description of
```

Load the message file and define the time at the start and end of trading hours as seconds after midnight.

```
TradeMessage = readmatrix(MSGFileName);
StartTradingSec   = 9.5*60*60;  % Starting time 9:30 in seconds after midnight
EndTradingSec     = 16*60*60;   % Ending time 16:00 in seconds after midnight
```

Define the trading interval as five seconds and pick message index values corresponding to the five-second intervals.

```
TradingIntervalSec = 5; % Trading interval length in seconds
IntervalBoundSec = (StartTradingSec:TradingIntervalSec:EndTradingSec)';
MessageSampleIdx = zeros(size(IntervalBoundSec));
k1 = 1;
for k2 = 1:size(TradeMessage,1)
    if TradeMessage(k2,1) >= IntervalBoundSec(k1)
        MessageSampleIdx(k1) = k2;
        k1 = k1+1;
    end
end
MessageSampleIdx(end) = length(TradeMessage);
```

Load and Prepare Limit Order Book Data

Load the limit order book data and define the number of levels in the limit order book.

```
LOB = readmatrix(LOBFileName);
NumLevels = 5;
```

Since LOBSTER stores prices in units of 10,000 dollars, convert the prices in the limit order book data into dollars by dividing by 10000.

```
LOB(:,1:2:(4*NumLevels)) = LOB(:,1:2:(4*NumLevels))./10000;
```

Using the message index values obtained previously, sample the limit order book at 5 second intervals.

```
SampledBook = LOB(MessageSampleIdx(2:end),1:NumLevels*4);
```

Define a one minute (60 seconds) trading horizon and compute the total number of intervals over 6.5 hours of trading data.

```
TradingHorizon = 1*60;
TotalNumIntervals = length(MessageSampleIdx(2:end));
TotalNumHorizons = 6.5*60*60/TradingHorizon;
```

The number of intervals per horizon is computed by dividing the total number of intervals by the total number of horizons.

```
NumIntervalsPerHorizon = round(TotalNumIntervals/TotalNumHorizons);
```

Allocate 75 percent of data for training and 25 percent for testing.

```
NumTrainingHorizons = round(TotalNumHorizons*0.75);
NumTestingHorizons = TotalNumHorizons - NumTrainingHorizons;
NumTrainingSteps = NumTrainingHorizons*NumIntervalsPerHorizon;
NumTestingSteps = NumTestingHorizons*NumIntervalsPerHorizon;
TrainingLOB = SampledBook(1:NumTrainingSteps,:);
TestingLOB = SampledBook(NumTrainingSteps+1:end,:);
```

Optimal Execution for Sell Trades

These sections compose the workflow for designing a DQN agent for the optimal execution of sell trades:

1. Define Settings for Sell Trades on page 4-392
2. TWAP Baseline Analysis on Training Data on page 4-392
3. Set Up RL Training Environment for Sell Trades on page 4-393
4. Validate Training Environment Reset and Step Functions for Sell Trades on page 4-394
5. Create and Train Agent for Sell Trades on page 4-396
6. Compute IS Using Trained Agent and Training Data for Sell Trades on page 4-401
7. Plot Training Results for Sell Trades on page 4-402
8. Display Summary of Training Results for Sell Trades on page 4-406
9. TWAP Baseline Analysis on Testing Data on page 4-407
10. Set Up RL Testing Environment for Sell Trades on page 4-408

11 Validate Testing Environment Reset and Step Functions for Sell Trades on page 4-409
12 Compute IS Using Trained Agent and Testing Data for Sell Trades on page 4-410
13 Plot Testing Results for Sell Trades on page 4-411
14 Display Summary of Training and Testing Results for Sell Trades on page 4-414

Define Settings for Sell Trades

To indicate sell trades, set the `BuyTrade` logical flag to `false`.

```
BuyTrade = false;
```

Define the total number of shares to sell over the horizon.

```
TotalTradingShares_Sell = 2738;
```

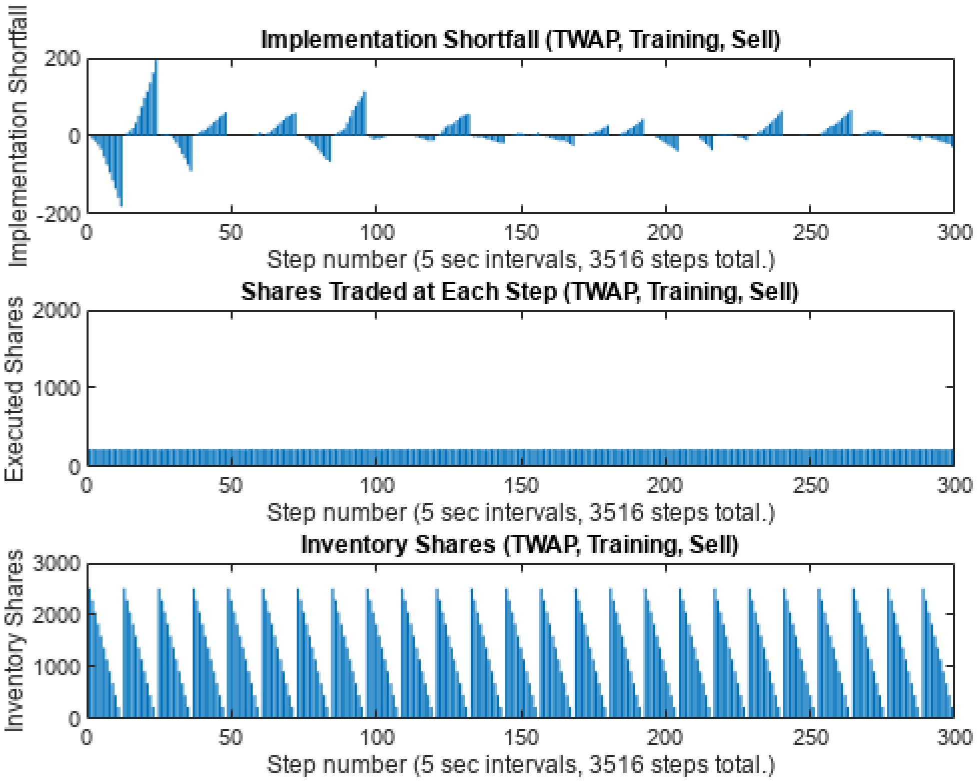
TWAP Baseline Analysis on Training Data

Execute TWAP trades and compute Implementation Shortfall (IS) using the `executeTWAPTrades` function, which uses the `tradeLOB` function to execute the trades and the `endingInventoryIS` function to compute IS. These functions are in the Local Functions on page 4-420 section.

```
[IS_TWAP_Horizon_Train_Sell,IS_TWAP_Step_Train_Sell, ...
    InventoryShares_Step_TWAP_Train_Sell,TradingShares_Step_TWAP_Train_Sell] = ...
    executeTWAPTrades(TrainingLOB,NumLevels,TotalTradingShares_Sell, ...
    NumIntervalsPerHorizon,NumTrainingHorizons,BuyTrade);
```

Plot the implementation shortfall, executed shares, and inventory shares for the TWAP baseline for each step using the training data.

```
figure
subplot(3,1,1)
bar(1:NumTrainingSteps, IS_TWAP_Step_Train_Sell)
xlabel(strcat("Step number (", num2str(TradingIntervalSec), ...
    " sec intervals, ", num2str(NumTrainingSteps), " steps total.)"))
ylabel("Implementation Shortfall")
title("Implementation Shortfall (TWAP, Training, Sell)")
xlim([0 300])

subplot(3,1,2)
bar(1:NumTrainingSteps, TradingShares_Step_TWAP_Train_Sell)
xlabel(strcat("Step number (", num2str(TradingIntervalSec), ...
    " sec intervals, ", num2str(NumTrainingSteps), " steps total.)"))
ylabel("Executed Shares")
title("Shares Traded at Each Step (TWAP, Training, Sell)")
xlim([0 300])
ylim([0 2000])

subplot(3,1,3)
bar(1:NumTrainingSteps, InventoryShares_Step_TWAP_Train_Sell)
xlabel(strcat("Step number (", num2str(TradingIntervalSec), ...
    " sec intervals, ", num2str(NumTrainingSteps), " steps total.)"))
ylabel("Inventory Shares")
title("Inventory Shares (TWAP, Training, Sell)")
xlim([0 300])
```



The TWAP executed shares in the middle panel have a flat profile, since the same number of shares are traded at each step. Also, the inventory shares in the bottom panel have a nearly perfect triangular sawtooth pattern, with the number of inventory shares declining linearly at each horizon as the shares are sold. The IS in the top panel sometimes declines to negative values (implying better executions than arrival price) for some horizons, while it increases to positive values (implying worse executions than arrival price) for other horizons.

Set Up RL Training Environment for Sell Trades

The RL environment requires observation and action information, reset function, and a step function.

Define the observation information for the four observation variables.

```
ObservationDimension = [1 4];
ObservationInfo_Train_Sell = rlNumericSpec(ObservationDimension);
ObservationInfo_Train_Sell.Name = 'Observation Data';
ObservationInfo_Train_Sell.Description = 'RemainingSharesToTrade, RemainingIntervals, IS_Agent, 
ObservationInfo_Train_Sell.LowerLimit = -inf;
ObservationInfo_Train_Sell.UpperLimit = inf;
```

Define the environment constants for the training data and the reset function. The function definition for `RL_OptimalExecution_LOB_ResetFcn` is in the Local Functions on page 4-420 section.

```
EnvConstants_Train_Sell.SampledBook = TrainingLOB;
EnvConstants_Train_Sell.BuyTrade = BuyTrade;
EnvConstants_Train_Sell.TotalTradingShares = TotalTradingShares_Sell;
EnvConstants_Train_Sell.NumIntervalsPerHorizon = NumIntervalsPerHorizon;
EnvConstants_Train_Sell.NumHorizons = NumTrainingHorizons;
EnvConstants_Train_Sell.NumSteps = NumTrainingSteps;
EnvConstants_Train_Sell.NumLevels = NumLevels;
EnvConstants_Train_Sell.IS_TWAP = IS_TWAP_Step_Train_Sell;

ResetFunction_Sell = @() RL_OptimalExecution_LOB_ResetFcn(EnvConstants_Train_Sell);
```

Define the action information as the number of shares to be traded at each step, ranging from zero shares to twice the number of shares as in TWAP.

```
NumActions = 39;
LowerActionRange = linspace(0,TradingShares_Step_TWAP_Train_Sell(1), round(NumActions/2));
ActionRange = round([LowerActionRange(1:end-1) ...
    linspace(TradingShares_Step_TWAP_Train_Sell(1), ...
    TradingShares_Step_TWAP_Train_Sell(1)*2,round(NumActions/2))]);

ActionInfo_Sell = rlFiniteSetSpec(ActionRange);
ActionInfo_Sell.Name = 'Trading Action (Number of Shares)';
ActionInfo_Sell.Description = 'Number of Shares to be Traded at Each Time Step';
```

Define the step function, which computes the rewards and updates the observations at each step. The function definition for `RL_OptimalExecution_LOB_StepFcn` is in the Local Functions on page 4-420 section.

```
StepFunction_Train_Sell = @(Action,LoggedSignals) ...
    RL_OptimalExecution_LOB_StepFcn(Action,LoggedSignals,EnvConstants_Train_Sell);
```

Create the environment using the custom function handles defined in this section.

```
RL_OptimalExecution_Training_Environment_Sell = rlFunctionEnv(ObservationInfo_Train_Sell,ActionIn
```

```
Reset!
Reset!

RL_OptimalExecution_Training_Environment_Sell = 
  rlFunctionEnv with properties:

         StepFcn: @(Action,LoggedSignals)RL_OptimalExecution_LOB_StepFcn(Action,LoggedSignals,En
        ResetFcn: @()RL_OptimalExecution_LOB_ResetFcn(EnvConstants_Train_Sell)
    LoggedSignals: [1×1 struct]
```

Validate Training Environment Reset and Step Functions for Sell Trades

Before using the created environment, the best practice is to validate the reset and step functions for the environment. Call the reset and step functions to see if they produce reasonable results without errors.

Validate the reset function and the initial observation.

```
InitialObservation = reset(RL_OptimalExecution_Training_Environment_Sell);
```

```
Reset!
```

```
InitialObservation
```

```
InitialObservation = 1×4
```

```
        2738          12           0           0
```

Validate the step function by monitoring the logged signals and rewards while calling the step functions multiple times. Here, you can see the time interval index increasing at each step and the current inventory shares changing as the trades are executed at each step.

```
[~,Reward1,~,LoggedSignals1] = step(RL_OptimalExecution_Training_Environment_Sell,TradingShares_S
LoggedSignals1
```

```
LoggedSignals1 = struct with fields:
               IntervalIdx: 1
                HorizonIdx: 1
    CurrentInventoryShares: 2624
              ArrivalPrice: 27.4600
            ExecutedShares: 114
            ExecutedPrices: 27.4600
     HorizonExecutedShares: [60×1 double]
     HorizonExecutedPrices: [60×1 double]
                  IS_Agent: [3516×1 double]
                    Reward: [3516×1 double]
    NumIntervalsPerHorizon: 12
               NumHorizons: 293
                 NumLevels: 5
```

```
Reward1
```

```
Reward1 = 0
```

```
[~,Reward2,~,LoggedSignals2] = step(RL_OptimalExecution_Training_Environment_Sell,10);
LoggedSignals2
```

```
LoggedSignals2 = struct with fields:
               IntervalIdx: 2
                HorizonIdx: 1
    CurrentInventoryShares: 2614
              ArrivalPrice: 27.4600
            ExecutedShares: 10
            ExecutedPrices: 27.4900
     HorizonExecutedShares: [60×1 double]
     HorizonExecutedPrices: [60×1 double]
                  IS_Agent: [3516×1 double]
                    Reward: [3516×1 double]
    NumIntervalsPerHorizon: 12
               NumHorizons: 293
                 NumLevels: 5
```

```
Reward2
```

```
Reward2 = 0.6150
```

```
[~,Reward3,~,LoggedSignals3] = step(RL_OptimalExecution_Training_Environment_Sell,35);
LoggedSignals3
```

```
LoggedSignals3 = struct with fields:
               IntervalIdx: 3
                HorizonIdx: 1
```

```
    CurrentInventoryShares: 2579
              ArrivalPrice: 27.4600
            ExecutedShares: 35
            ExecutedPrices: 27.4900
     HorizonExecutedShares: [60×1 double]
     HorizonExecutedPrices: [60×1 double]
                  IS_Agent: [3516×1 double]
                    Reward: [3516×1 double]
    NumIntervalsPerHorizon: 12
               NumHorizons: 293
                 NumLevels: 5
```

```
Reward3
```

```
Reward3 = 0.8925
```

Call the reset function again to restore the environment back to the initial state.

```
reset(RL_OptimalExecution_Training_Environment_Sell);
```

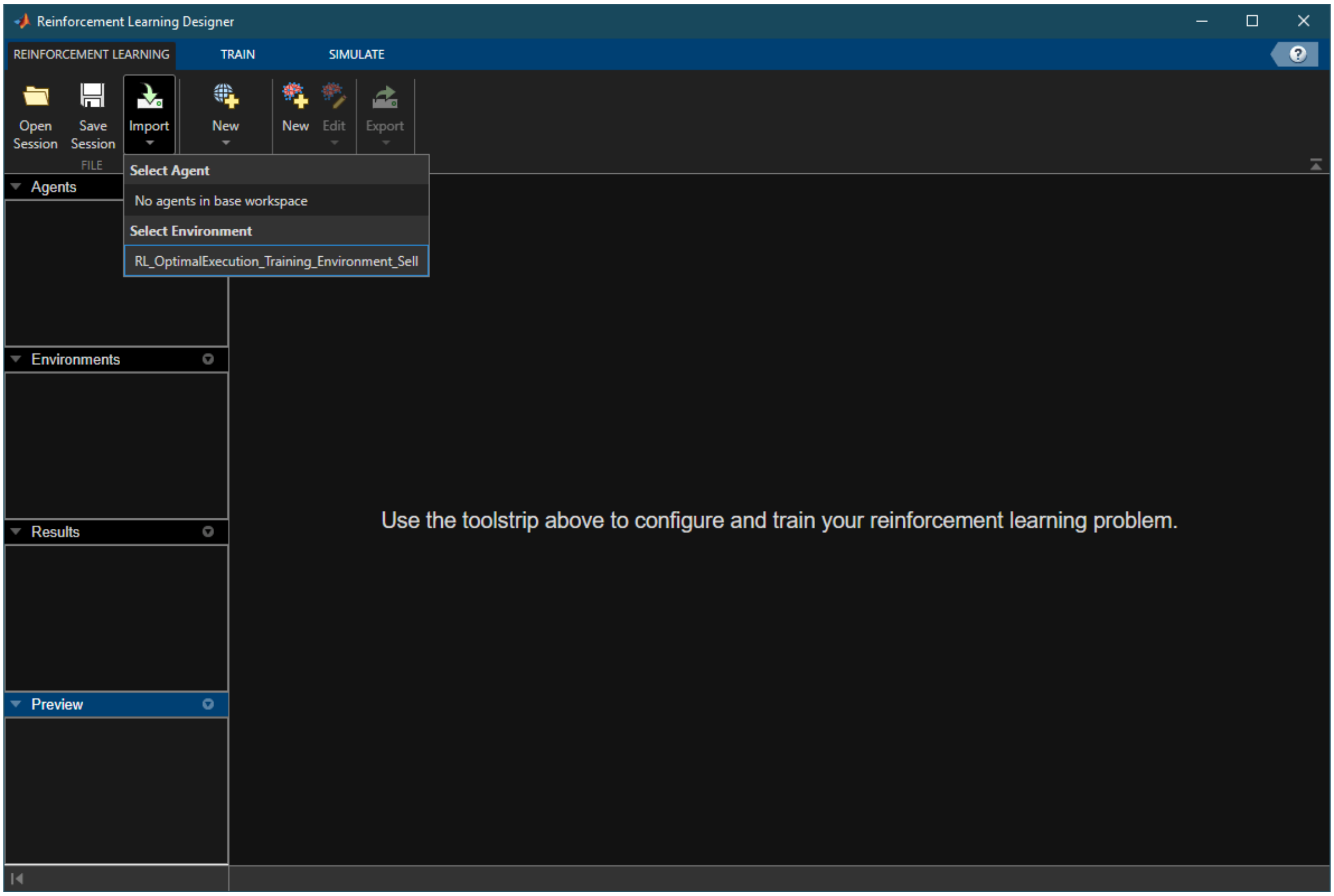
```
Reset!
```

Create and Train Agent for Sell Trades

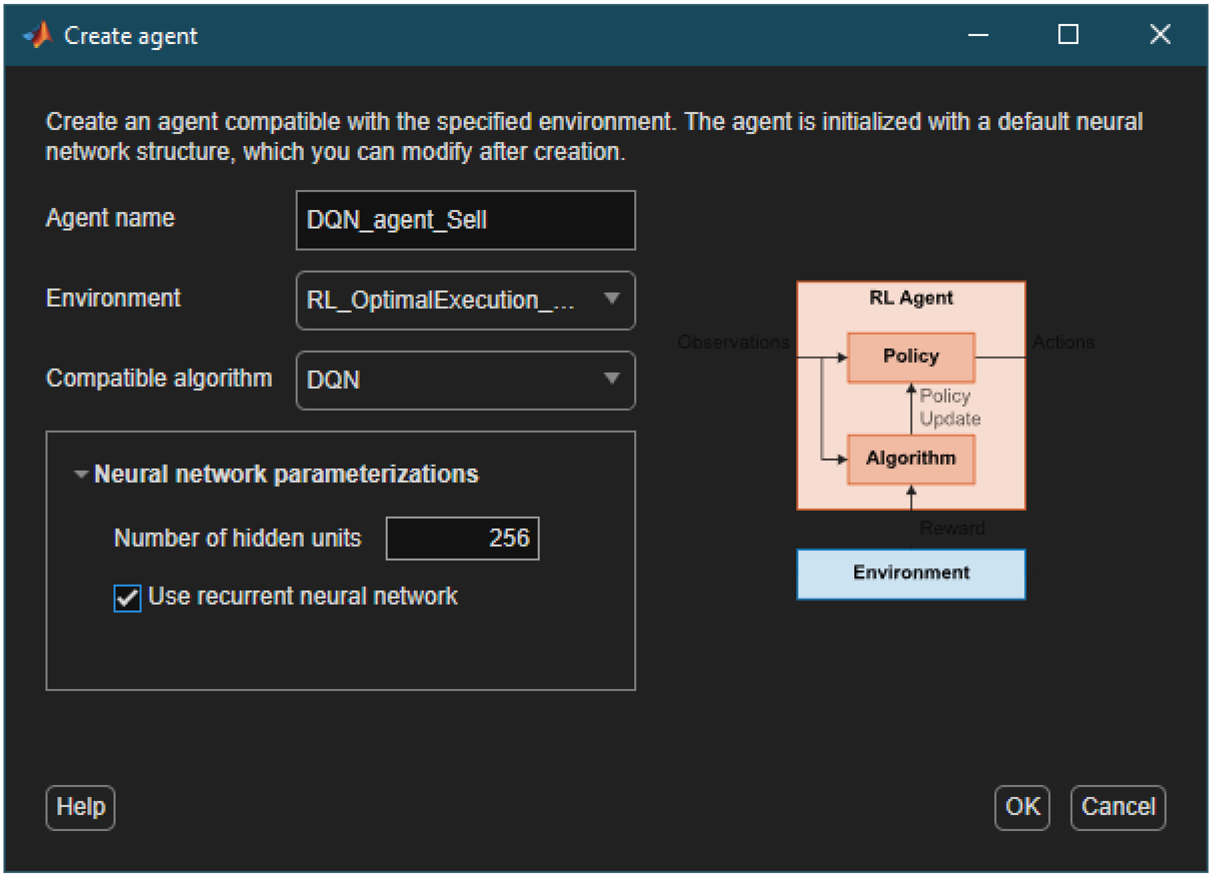
Open Reinforcement Learning Designer (Reinforcement Learning Toolbox).

```
reinforcementLearningDesigner
```

Import the training environment by clicking the **Import** button and selecting the created training environment.


Reinforcement Learning Designer
REINFORCEMENT LEARNING
TRAIN
SIMULATE
Open Session
Save Session
Import
New
New
Edit
Export
FILE
Select Agent
No agents in base workspace
Select Environment
RL_OptimalExecution_Training_Environment_Sell
Agents
Environments
Results
Preview
Use the toolstrip above to configure and train your reinforcement learning problem.

Create a new DQN agent by selecting **New > Agent**, then selecting `DQN` from the **Compatible algorithm** menu. Enable LSTM by selecting **Use recurrent neural network**.



Create an agent compatible with the specified environment. The agent is initialized with a default neural network structure, which you can modify after creation.

Agent name: DQN_agent_Sell

Environment: RL_OptimalExecution_...

Compatible algorithm: DQN

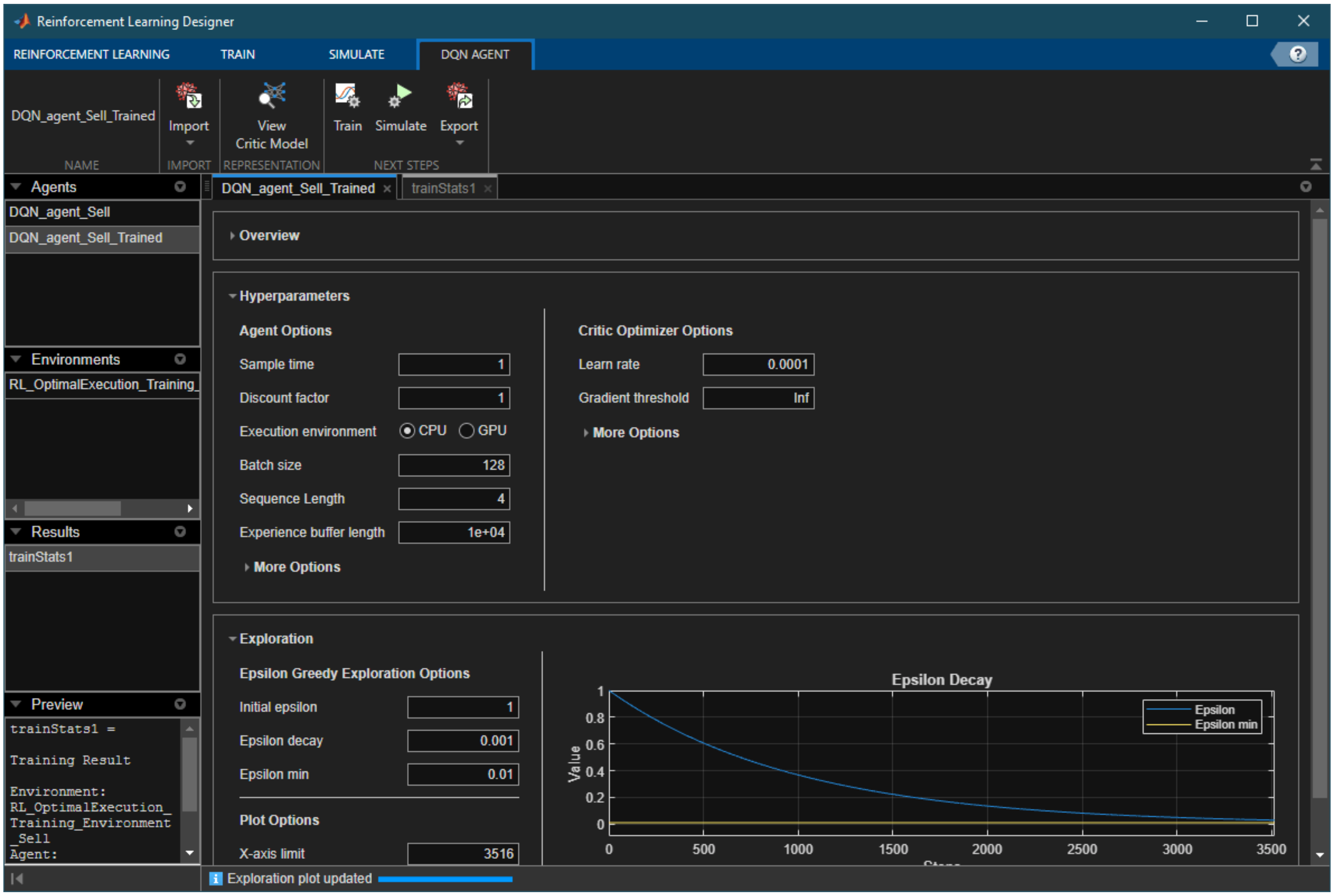
Neural network parameterizations

Number of hidden units: 256

☑ Use recurrent neural network

Set the agent hyperparameters and exploration parameters in the **DQN Agent** tab. Use the default options, except for the following:

- **Discount factor** — 1
- **Learn rate** — `0.0001`
- **Batch size** — 128
- **Sequence length** — 4
- **Initial epsilon** — 1
- **Epsilon decay** — `0.001`



```
trainStats1 =

Training Result

Environment:
RL_OptimalExecution_
Training_Environment
_Sell
Agent:
```

Agent Options

Sample time	1
Discount factor	1
Execution environment	● CPU ○ GPU
Batch size	128
Sequence Length	4
Experience buffer length	1e+04

Critic Optimizer Options

Learn rate	0.0001
Gradient threshold	Inf

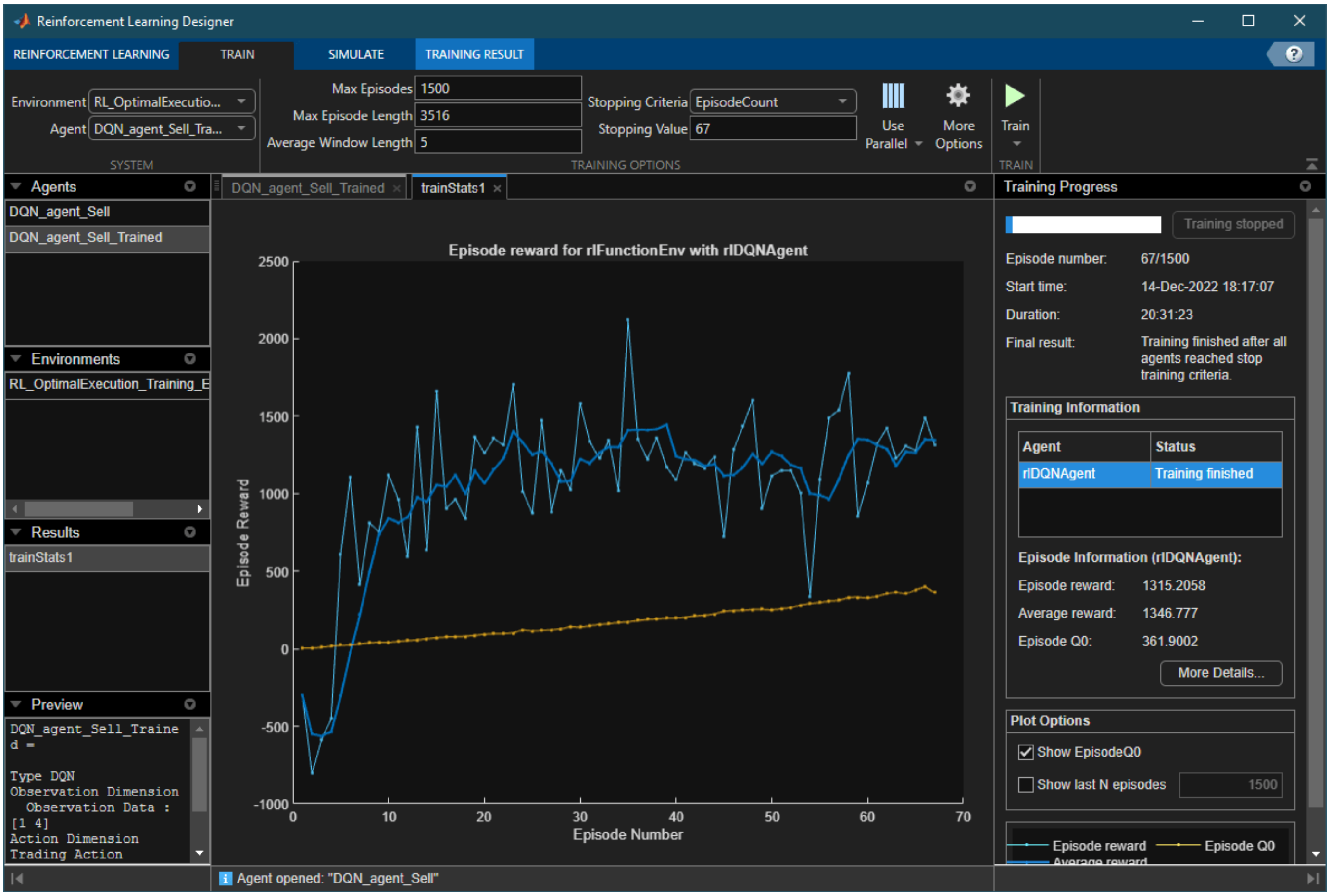
Epsilon Greedy Exploration Options

Initial epsilon	1
Epsilon decay	0.001
Epsilon min	0.01

Plot Options

X-axis limit	3516

Go to the **Train** tab, and set the **Max Episode Length** to `3516` (`NumTrainingSteps`). You can experiment with different **Stopping Criteria**, or you can manually stop the training by clicking on the **Stop** button. In this example, you set the **Stopping Criteria** to `EpisodeCount` and the **Max Episodes** to a value greater than or equal to `EpisodeCount`. Click **Train** to start the training.



Reinforcement Learning Designer

REINFORCEMENT LEARNING | TRAIN | SIMULATE | TRAINING RESULT

SYSTEM: Environment: RL_OptimalExecutio... Agent: DQN_agent_Sell_Tra...

TRAINING OPTIONS: Max Episodes: 1500; Max Episode Length: 3516; Average Window Length: 5; Stopping Criteria: EpisodeCount; Stopping Value: 67

Agents: DQN_agent_Sell, DQN_agent_Sell_Trained

Environments: RL_OptimalExecution_Training_E

Results: trainStats1

Preview:

```
DQN_agent_Sell_Traine
d =

Type DQN
Observation Dimension
  Observation Data :
[1 4]
Action Dimension
Trading Action
```

Agent opened: "DQN_agent_Sell"

Training Progress

Training stopped

Episode number: 67/1500
Start time: 14-Dec-2022 18:17:07
Duration: 20:31:23
Final result: Training finished after all agents reached stop training criteria.

Training Information

Agent	Status
rlDQNAgent	Training finished

Episode Information (rlDQNAgent):
Episode reward: 1315.2058
Average reward: 1346.777
Episode Q0: 361.9002

Plot Options
☑ Show EpisodeQ0
□ Show last N episodes 1500

Once the training is complete, export the trained agent to the workspace. Since training can take a long time, you can load a pretrained agent (DQN_agent_Sell_Trained.mat) for sell trades.

```
load DQN_agent_Sell_Trained.mat
```

You can view the DQN agent options by examining the AgentOptions property.

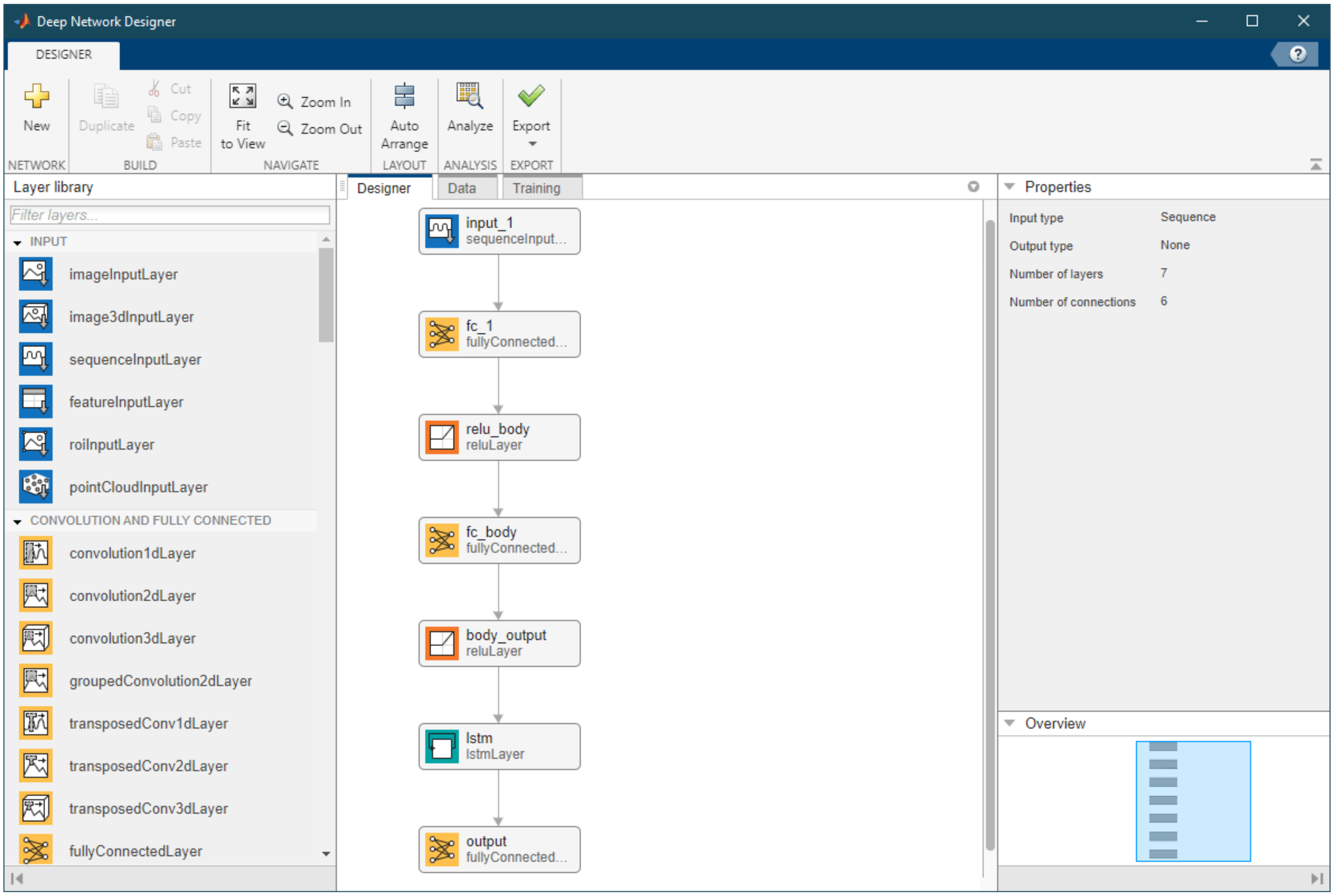
```
DQN_agent_Sell_Trained.AgentOptions
```

```
ans = 
  rlDQNAgentOptions with properties:

                           UseDoubleDQN: 1
               EpsilonGreedyExploration: [1×1 rl.option.EpsilonGreedyExploration]
                 CriticOptimizerOptions: [1×1 rl.option.rlOptimizerOptions]
            BatchDataRegularizerOptions: []
                     TargetSmoothFactor: 1.0000e-03
                  TargetUpdateFrequency: 1
    ResetExperienceBufferBeforeTraining: 0
                         SequenceLength: 4
                          MiniBatchSize: 128
                    NumStepsToLookAhead: 1
                 ExperienceBufferLength: 10000
                             SampleTime: 1
                         DiscountFactor: 1
```

```
        InfoToSave: [1×1 struct]
```

Also, you can visualize the DQN agent's critic network by using Deep Network Designer (Deep Learning Toolbox).

```
deepNetworkDesigner(layerGraph(getModel(getCritic(DQN_agent_Sell_Trained))))
```



Input type	Sequence
Output type	None
Number of layers	7
Number of connections	6

Compute IS Using Trained Agent and Training Data for Sell Trades

Simulate selling actions using the trained agent and training environment.

```
rng('default');
reset(RL_OptimalExecution_Training_Environment_Sell);
```

```
Reset!
```

```
simOptions = rlSimulationOptions(MaxSteps=NumTrainingSteps)
```

```
simOptions = 
  rlSimulationOptions with properties:

                  MaxSteps: 3516
            NumSimulations: 1
               StopOnError: "on"
               UseParallel: 0
```

```
    ParallelizationOptions: [1×1 rl.option.ParallelSimulation]
```

```
Trained_Agent_Sell = DQN_agent_Sell_Trained;
experience_Sell = sim(RL_OptimalExecution_Training_Environment_Sell,Trained_Agent_Sell,simOptions
```

```
Reset!
Last Horizon. Episode is Done!
HorizonIdx:

ans = 293
```

```
SimAction_Train_Sell = squeeze(experience_Sell.Action.TradingAction_NumberOfShares_.Data);
```

Compute the IS using simulated actions on the training data.

```
NumSimSteps = length(SimAction_Train_Sell);
SimInventory_Train_Sell = nan(NumSimSteps,1);
SimExecutedShares_Train_Sell = SimInventory_Train_Sell;
SimReward_Train_Sell = nan(NumSimSteps,1);

reset(RL_OptimalExecution_Training_Environment_Sell);
```

```
Reset!
```

```
for k=1:NumSimSteps
    [~,SimReward_Train_Sell(k),~,LoggedSignals] = ...
        step(RL_OptimalExecution_Training_Environment_Sell,SimAction_Train_Sell(k));
    SimInventory_Train_Sell(k) = LoggedSignals.CurrentInventoryShares;
    SimExecutedShares_Train_Sell(k) = sum(LoggedSignals.ExecutedShares);
end
```

```
Last Horizon. Episode is Done!
HorizonIdx:

ans = 293
```

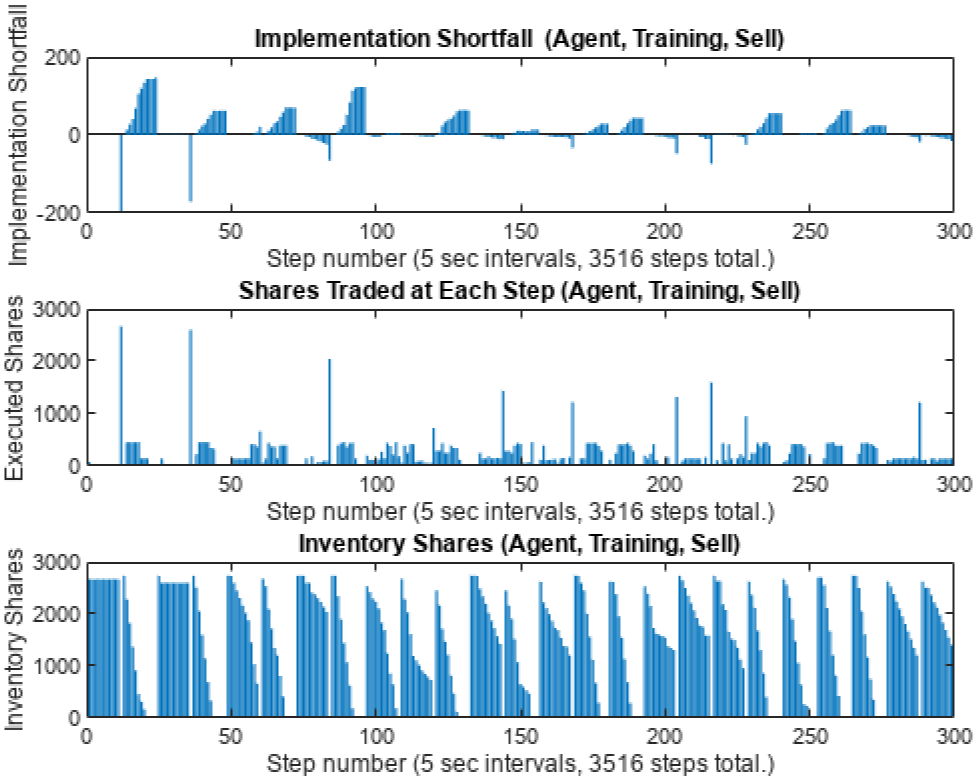
```
IS_Agent_Step_Train_Sell = LoggedSignals.IS_Agent;
IS_Agent_Horizon_Train_Sell = IS_Agent_Step_Train_Sell( ....
    NumIntervalsPerHorizon:NumIntervalsPerHorizon:NumTrainingSteps);
```

Plot Training Results for Sell Trades

Plot Implementation Shortfall (IS), executed shares, and inventory shares for the agent for each step using training data.

```
figure
subplot(3,1,1)
bar(1:NumTrainingSteps, IS_Agent_Step_Train_Sell)
xlabel(strcat("Step number (", num2str(TradingIntervalSec), ...
    " sec intervals, ", num2str(NumTrainingSteps), " steps total.)"))
ylabel("Implementation Shortfall")
title("Implementation Shortfall  (Agent, Training, Sell)")
xlim([0 300])
ylim([-200 200])

subplot(3,1,2)
bar(1:NumTrainingSteps, SimExecutedShares_Train_Sell)
xlabel(strcat("Step number (", num2str(TradingIntervalSec), ...
    " sec intervals, ", num2str(NumTrainingSteps), " steps total.)"))
```

```
ylabel("Executed Shares")
title("Shares Traded at Each Step (Agent, Training, Sell)")
xlim([0 300])

subplot(3,1,3)
bar(1:NumTrainingSteps, SimInventory_Train_Sell)
xlabel(strcat("Step number (", num2str(TradingIntervalSec), ...
    " sec intervals, ", num2str(NumTrainingSteps), " steps total.)"))
ylabel("Inventory Shares")
title("Inventory Shares (Agent, Training, Sell)")
xlim([0 300])
```

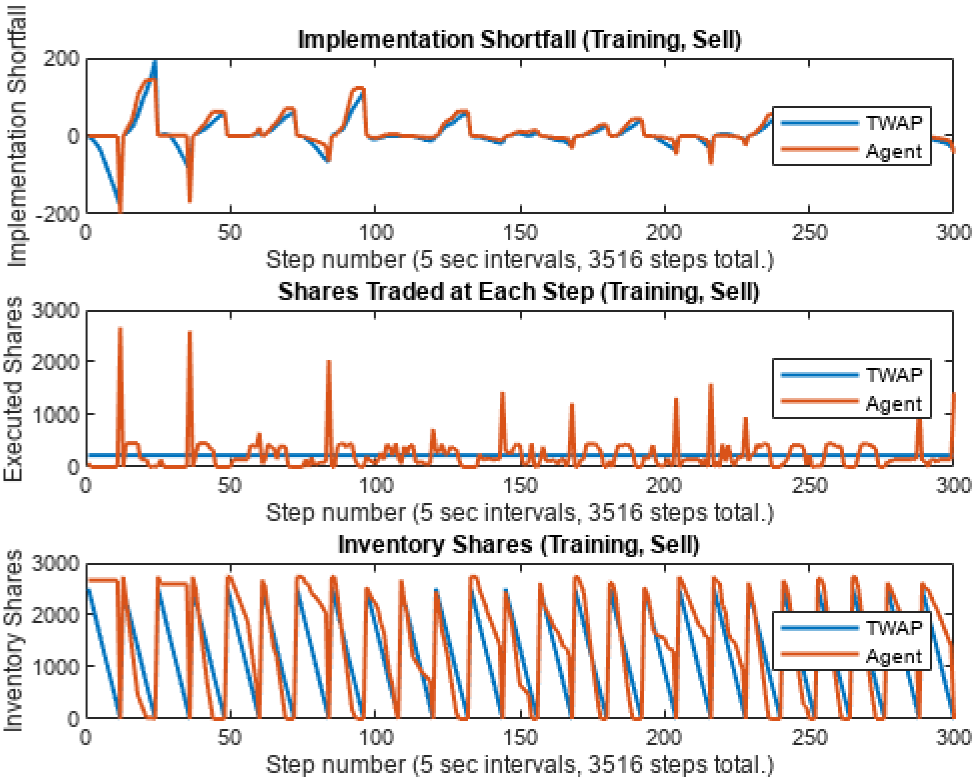


Unlike the previous TWAP baseline plot, where the executed shares in the middle panel have a flat profile, here you can see that the agent actively changes the number of executed shares at each step. Also, the inventory shares in the bottom panel decline in a non-linear manner.

To directly compare the results from the agent with the TWAP baseline, you can plot both results on the same chart. Generate plots to compare implementation shortfall, executed shares, and inventory shares between the TWAP baseline and the agent for each step using the training data.

```
figure
subplot(3,1,1)
plot(1:NumTrainingSteps, [IS_TWAP_Step_Train_Sell IS_Agent_Step_Train_Sell], linewidth=1.5)
xlabel(strcat("Step number (", num2str(TradingIntervalSec), ...
    " sec intervals, ", num2str(NumTrainingSteps), " steps total.)"))
ylabel("Implementation Shortfall")
title("Implementation Shortfall (Training, Sell)")
```

```
legend(["TWAP" "Agent"], location="east")
xlim([0 300])

subplot(3,1,2)
plot(1:NumTrainingSteps, [TradingShares_Step_TWAP_Train_Sell ...
    SimExecutedShares_Train_Sell], linewidth=1.5)
xlabel(strcat("Step number (", num2str(TradingIntervalSec), ...
    " sec intervals, ", num2str(NumTrainingSteps), " steps total.)"))
ylabel("Executed Shares")
title("Shares Traded at Each Step (Training, Sell)")
legend(["TWAP" "Agent"], location="east")
xlim([0 300])

subplot(3,1,3)
plot(1:NumTrainingSteps, [InventoryShares_Step_TWAP_Train_Sell ...
    SimInventory_Train_Sell], linewidth=1.5)
xlabel(strcat("Step number (", num2str(TradingIntervalSec), ...
    " sec intervals, ", num2str(NumTrainingSteps), " steps total.)"))
ylabel("Inventory Shares")
title("Inventory Shares (Training, Sell)")
legend(["TWAP" "Agent"], location="east")
xlim([0 300])
```



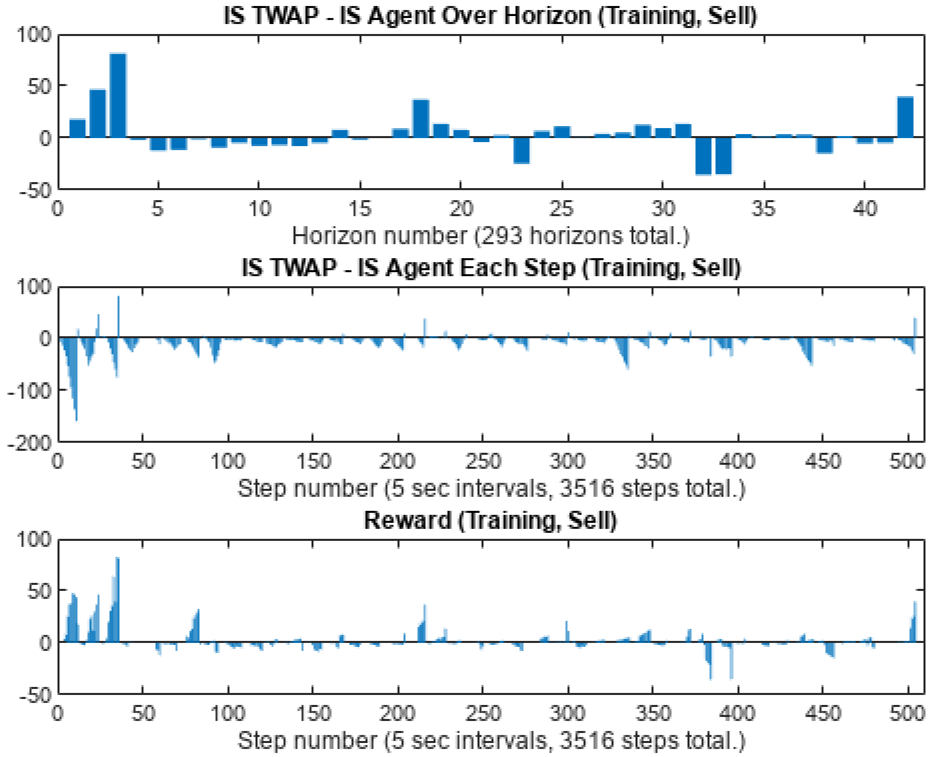
You can see the differences between the TWAP baseline (blue) and the agent (red) in their IS, executed shares, and inventory shares. Also, the agent tends to behave differently when the IS trends negative as compared to positive.

- For example, in the first horizon (step numbers 1 to 12) and the third horizon (step numbers 25 to 36), the IS trends negative (more favorable prices than the arrival price) and the agent initially trades fewer shares than TWAP, until there are large spikes in the executed shares at the ends of the horizons to meet the ending inventory constraint. Also, the inventories initially declines more slowly than TWAP, until the abrupt declines to zero at the end of the horizon. This coincided with negative pulls in the agent's IS at the end of the first and the third horizons that go even lower than the IS of TWAP.
- On the other hand, in the second horizon (step numbers 13 to 24), the IS trends positive (less favorable prices than the arrival price), and the agent initially trades more shares than TWAP to liquidate the inventory faster than TWAP. This has a limiting effect on the agent's IS at the end of the horizon, which is also lower than the IS of TWAP.

You can see similar behaviors in other horizons, although they do not always lead to an outperformance of the agent over TWAP.

Also, you can plot the differences in IS between TWAP and the agent, and then compare them with the rewards.

```
figure
subplot(3,1,1)
bar(1:NumTrainingHorizons, IS_TWAP_Horizon_Train_Sell - IS_Agent_Horizon_Train_Sell)
xlabel(strcat("Horizon number (", num2str(NumTrainingHorizons), " horizons total.)"))
title("IS TWAP - IS Agent Over Horizon (Training, Sell)")
xlim([0 43])
subplot(3,1,2)
bar(1:NumTrainingSteps, IS_TWAP_Step_Train_Sell - IS_Agent_Step_Train_Sell)
xlabel(strcat("Step number (", num2str(TradingIntervalSec), ...
    " sec intervals, ", num2str(NumTrainingSteps), " steps total.)"))
title("IS TWAP - IS Agent Each Step (Training, Sell)")
xlim([0 510])
subplot(3,1,3)
bar(1:NumTrainingSteps, SimReward_Train_Sell)
xlabel(strcat("Step number (", num2str(TradingIntervalSec), ...
    " sec intervals, ", num2str(NumTrainingSteps), " steps total.)"))
title("Reward (Training, Sell)")
xlim([0 510])
```



The top panel ($\mathrm{IS}_{\mathrm{TWAP}}(0, T) - \mathrm{IS}_{\mathrm{Agent}}(0, T)$) shows the final performance of the agent for the training data as the differences in IS between TWAP and the agent at the ends of the horizons at times T. For example, in the top panel, you can see the outperformance of the agent (positive IS difference) in the first three horizons, while the agent underperforms in some of the other horizons. The middle panel ($\mathrm{IS}_{\mathrm{TWAP}}(0, t) - \mathrm{IS}_{\mathrm{Agent}}(0, t)$) shows the differences in IS for all time steps t, including the ones before the ends of the horizons. The middle panel does not appear to track the top panel values very well in some steps (except for the very last steps in the horizons, which are the top panel values when $t = T$), as there are frequent trend reversals between negative and positive values at the last steps of the horizons. Therefore, the simple difference in IS between TWAP and the agent at each step ($\mathrm{IS}_{\mathrm{TWAP}}(0, t) - \mathrm{IS}_{\mathrm{Agent}}(0, t)$) shown in the middle panel may not be a good reward function. On the other hand, the bottom panel, which plots the rewards used by the agent in this example at each step, appears to track the top panel values more closely than the middle panel at most steps.

Display Summary of Training Results for Sell Trades

Display the total Implementation Shortfall outperformance of the agent over TWAP for the training data.

```
sum(IS_TWAP_Horizon_Train_Sell - IS_Agent_Horizon_Train_Sell)
```

```
ans = 164.4700
```

Display total-gain-to-loss ratio (TGLR) for the training data.

```
ISTGLR(IS_TWAP_Horizon_Train_Sell - IS_Agent_Horizon_Train_Sell)
```

```
ans = 1.2327
```

TWAP Baseline Analysis on Testing Data

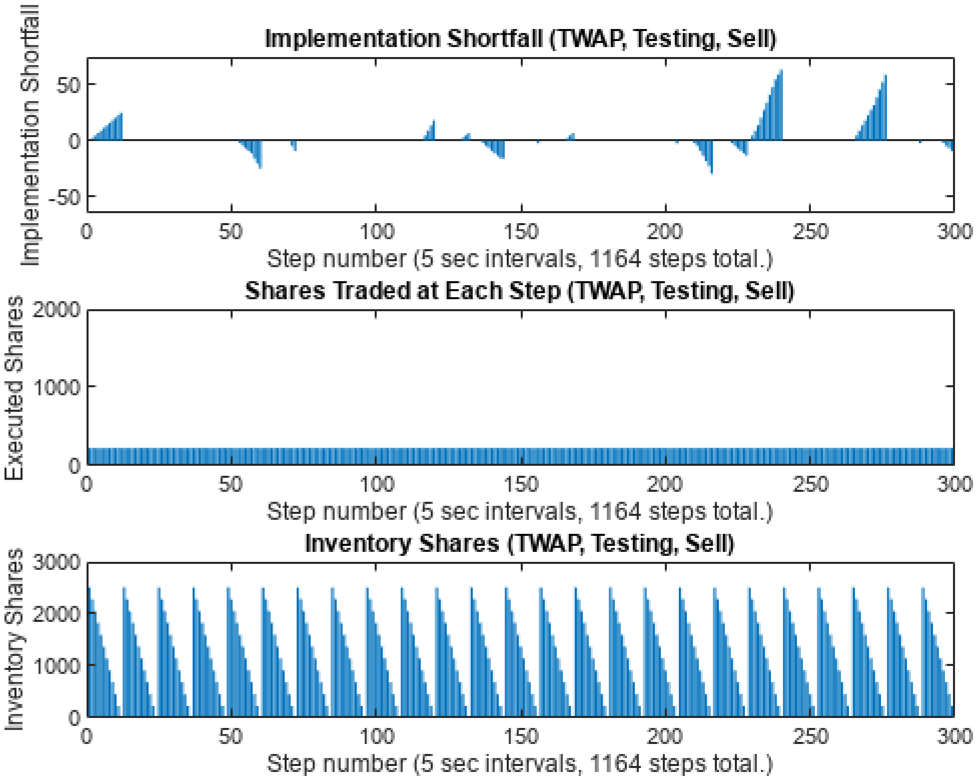
Using the same trained agent as the Compute IS Using Trained Agent and Training Data for Sell Trades on page 4-401 section, you can perform the same analysis on the testing data that you kept separate from the training data.

First, execute the TWAP trades and compute IS for the testing data.

```
[IS_TWAP_Horizon_Test_Sell,IS_TWAP_Step_Test_Sell, ...
    InventoryShares_Step_TWAP_Test_Sell,TradingShares_Step_TWAP_Test_Sell] = ...
    executeTWAPTrades(TestingLOB,NumLevels,TotalTradingShares_Sell,...
    NumIntervalsPerHorizon,NumTestingHorizons,BuyTrade);
```

Plot the implementation shortfall, executed shares, and inventory shares for the TWAP baseline for each step using testing data.

```
figure
subplot(3,1,1)
bar(1:NumTestingSteps, IS_TWAP_Step_Test_Sell)
xlabel(strcat("Step number (", num2str(TradingIntervalSec), ...
    " sec intervals, ", num2str(NumTestingSteps), " steps total.)"))
ylabel("Implementation Shortfall")
title("Implementation Shortfall (TWAP, Testing, Sell)")
xlim([0 300])
ylim([-65 75])

subplot(3,1,2)
bar(1:NumTestingSteps, TradingShares_Step_TWAP_Test_Sell)
xlabel(strcat("Step number (", num2str(TradingIntervalSec), ...
    " sec intervals, ", num2str(NumTestingSteps), " steps total.)"))
ylabel("Executed Shares")
title("Shares Traded at Each Step (TWAP, Testing, Sell)")
xlim([0 300])
ylim([0 2000])

subplot(3,1,3)
bar(1:NumTestingSteps, InventoryShares_Step_TWAP_Test_Sell)
xlabel(strcat("Step number (", num2str(TradingIntervalSec), ...
    " sec intervals, ", num2str(NumTestingSteps), " steps total.)"))
ylabel("Inventory Shares")
title("Inventory Shares (TWAP, Testing, Sell)")
xlim([0 300])
```



Set Up RL Testing Environment for Sell Trades

Next, define the reset function for the testing environment.

```
EnvConstants_Test_Sell.SampledBook = TestingLOB;
EnvConstants_Test_Sell.BuyTrade = BuyTrade;
EnvConstants_Test_Sell.TotalTradingShares = TotalTradingShares_Sell;
EnvConstants_Test_Sell.NumIntervalsPerHorizon = NumIntervalsPerHorizon;
EnvConstants_Test_Sell.NumHorizons = NumTestingHorizons;
EnvConstants_Test_Sell.NumSteps = NumTestingSteps;
EnvConstants_Test_Sell.NumLevels = NumLevels;
EnvConstants_Test_Sell.IS_TWAP = IS_TWAP_Step_Test_Sell;

ResetFunction_Test_Sell = @() RL_OptimalExecution_LOB_ResetFcn(EnvConstants_Test_Sell);
```

Define the step function for the testing environment.

```
StepFunction_Test_Sell = @(Action,LoggedSignals) ...
    RL_OptimalExecution_LOB_StepFcn(Action,LoggedSignals,EnvConstants_Test_Sell);
```

Create the testing environment using the reset and step function handles.

```
RL_OptimalExecution_Testing_Environment_Sell = ...
    rlFunctionEnv(ObservationInfo_Train_Sell,ActionInfo_Sell, ...
    StepFunction_Test_Sell,ResetFunction_Test_Sell)
```

```
Reset!
Reset!
```

```
RL_OptimalExecution_Testing_Environment_Sell = 
  rlFunctionEnv with properties:

         StepFcn: @(Action,LoggedSignals)RL_OptimalExecution_LOB_StepFcn(Action,LoggedSignals,En
        ResetFcn: @()RL_OptimalExecution_LOB_ResetFcn(EnvConstants_Test_Sell)
    LoggedSignals: [1×1 struct]
```

Validate Testing Environment Reset and Step Functions for Sell Trades

Now, validate the testing environment using reset and step function handles.

```
InitialObservation = reset(RL_OptimalExecution_Testing_Environment_Sell);
```

```
Reset!
```

```
InitialObservation
```

```
InitialObservation = 1×4

        2738          12           0           0
```

```
[~,Reward1,~,LoggedSignals1] = step( ...
    RL_OptimalExecution_Testing_Environment_Sell,TradingShares_Step_TWAP_Train_Sell(1));
LoggedSignals1
```

```
LoggedSignals1 = struct with fields:
               IntervalIdx: 1
                HorizonIdx: 1
    CurrentInventoryShares: 2624
              ArrivalPrice: 26.8000
            ExecutedShares: 114
            ExecutedPrices: 26.8000
     HorizonExecutedShares: [60×1 double]
     HorizonExecutedPrices: [60×1 double]
                  IS_Agent: [1164×1 double]
                    Reward: [1164×1 double]
    NumIntervalsPerHorizon: 12
               NumHorizons: 97
                 NumLevels: 5
```

```
Reward1
```

```
Reward1 = 0
```

```
[~,Reward2,~,LoggedSignals2] = step( ...
    RL_OptimalExecution_Testing_Environment_Sell,10);
LoggedSignals2
```

```
LoggedSignals2 = struct with fields:
               IntervalIdx: 2
                HorizonIdx: 1
    CurrentInventoryShares: 2614
              ArrivalPrice: 26.8000
            ExecutedShares: 10
            ExecutedPrices: 26.7900
     HorizonExecutedShares: [60×1 double]
     HorizonExecutedPrices: [60×1 double]
```

```
                  IS_Agent: [1164×1 double]
                    Reward: [1164×1 double]
    NumIntervalsPerHorizon: 12
               NumHorizons: 97
                 NumLevels: 5
```

```
Reward2
```

```
Reward2 = -0.2050
```

```
[~,Reward3,~,LoggedSignals3] = step( ...
    RL_OptimalExecution_Testing_Environment_Sell,35);
LoggedSignals3
```

```
LoggedSignals3 = struct with fields:
               IntervalIdx: 3
                HorizonIdx: 1
    CurrentInventoryShares: 2579
              ArrivalPrice: 26.8000
            ExecutedShares: 35
            ExecutedPrices: 26.7900
     HorizonExecutedShares: [60×1 double]
     HorizonExecutedPrices: [60×1 double]
                  IS_Agent: [1164×1 double]
                    Reward: [1164×1 double]
    NumIntervalsPerHorizon: 12
               NumHorizons: 97
                 NumLevels: 5
```

```
Reward3
```

```
Reward3 = -0.2975
```

Call the reset function again to restore the environment back to the initial state.

```
reset(RL_OptimalExecution_Testing_Environment_Sell);
```

```
Reset!
```

Compute IS Using Trained Agent and Testing Data for Sell Trades

Simulate actions using the trained agent and testing data for sell trades.

```
simOptions = rlSimulationOptions(MaxSteps=NumTestingSteps);
experience_Sell = sim(RL_OptimalExecution_Testing_Environment_Sell,Trained_Agent_Sell,simOptions)
```

```
Reset!
Last Horizon. Episode is Done!
HorizonIdx:

ans = 97
```

```
SimAction_Test_Sell = squeeze(experience_Sell.Action.TradingAction_NumberOfShares_.Data);
```

Compute Implementation Shortfall (IS) using simulated actions on testing data.

```
NumSimSteps = length(SimAction_Test_Sell);
SimInventory_Test_Sell = nan(NumSimSteps,1);
```

```
SimExecutedShares_Test_Sell = SimInventory_Test_Sell;
SimReward_Test_Sell = nan(NumSimSteps,1);

reset(RL_OptimalExecution_Testing_Environment_Sell);
```

```
Reset!
```

```
for k=1:NumSimSteps
    [~,SimReward_Test_Sell(k),~,LoggedSignals] = step( ...
        RL_OptimalExecution_Testing_Environment_Sell,SimAction_Test_Sell(k));
    SimInventory_Test_Sell(k) = LoggedSignals.CurrentInventoryShares;
    SimExecutedShares_Test_Sell(k) = sum(LoggedSignals.ExecutedShares);
end
```

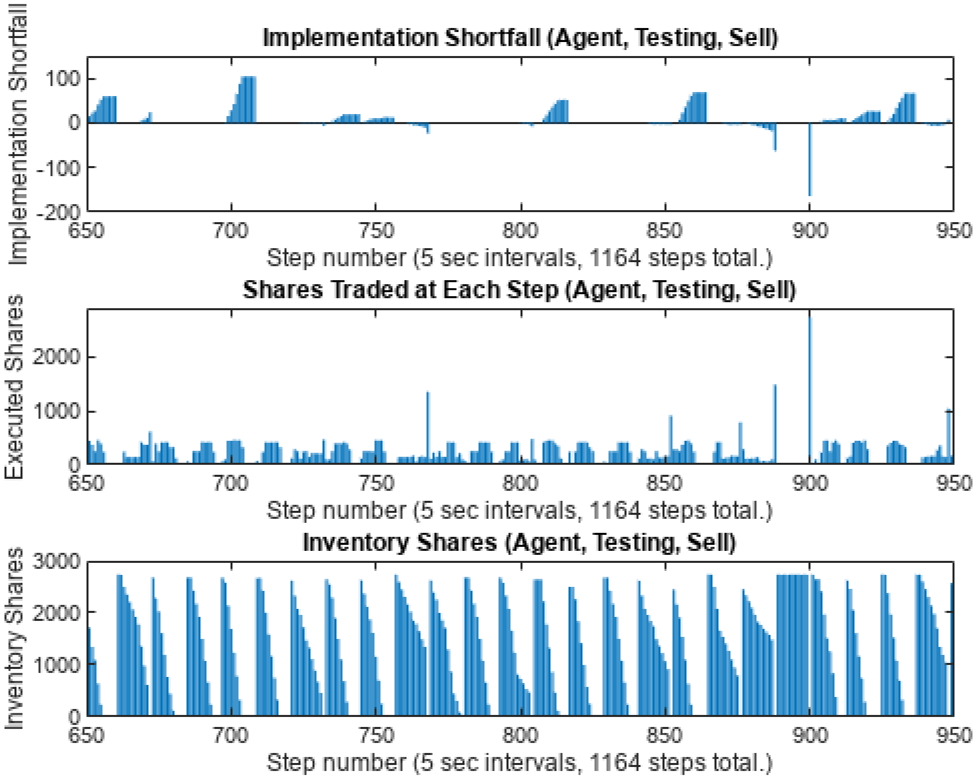
```
Last Horizon. Episode is Done!
HorizonIdx:

ans = 97
```

```
IS_Agent_Step_Test_Sell = LoggedSignals.IS_Agent;
IS_Agent_Horizon_Test_Sell = IS_Agent_Step_Test_Sell( ...
    NumIntervalsPerHorizon:NumIntervalsPerHorizon:NumTestingSteps);
```

Plot Testing Results for Sell Trades

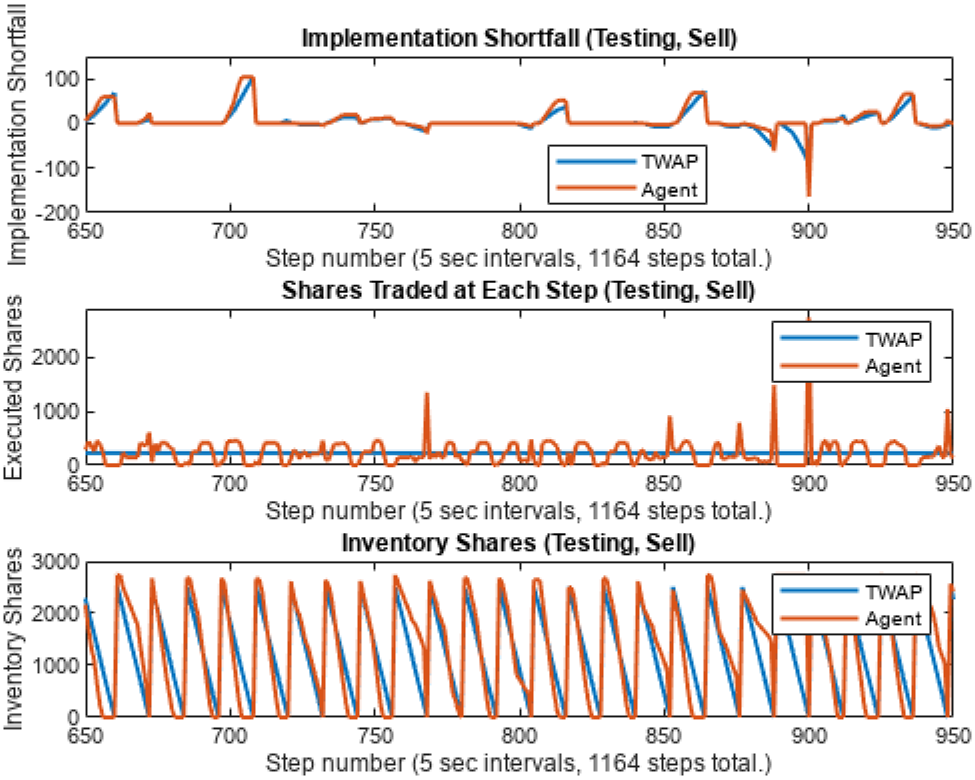
Plot Implementation Shortfall (IS), executed shares, and inventory shares of the agent for each step using testing data.

```
figure
subplot(3,1,1)
bar(1:NumTestingSteps, IS_Agent_Step_Test_Sell)
xlabel(strcat("Step number (", num2str(TradingIntervalSec), ...
    " sec intervals, ", num2str(NumTestingSteps), " steps total.)"))
ylabel("Implementation Shortfall")
title("Implementation Shortfall (Agent, Testing, Sell)")
xlim([650 950])
ylim([-200 150])

subplot(3,1,2)
bar(1:NumTestingSteps, SimExecutedShares_Test_Sell)
xlabel(strcat("Step number (", num2str(TradingIntervalSec), ...
    " sec intervals, ", num2str(NumTestingSteps), " steps total.)"))
ylabel("Executed Shares")
title("Shares Traded at Each Step (Agent, Testing, Sell)")
xlim([650 950])
ylim([0 2900])

subplot(3,1,3)
bar(1:NumTestingSteps, SimInventory_Test_Sell)
xlabel(strcat("Step number (", num2str(TradingIntervalSec), ...
    " sec intervals, ", num2str(NumTestingSteps), " steps total.)"))
ylabel("Inventory Shares")
title("Inventory Shares (Agent, Testing, Sell)")
xlim([650 950])
```



Compare implementation shortfall, executed shares, and inventory shares between the TWAP baseline and the agent for each step using testing data.

```
figure
subplot(3,1,1)
plot(1:NumTestingSteps, [IS_TWAP_Step_Test_Sell IS_Agent_Step_Test_Sell], linewidth=1.5)
xlabel(strcat("Step number (", num2str(TradingIntervalSec), ...
    " sec intervals, ", num2str(NumTestingSteps), " steps total.)"))
ylabel("Implementation Shortfall")
title("Implementation Shortfall (Testing, Sell)")
legend(["TWAP" "Agent"],location='best')
xlim([650 950])
ylim([-200 150])

subplot(3,1,2)
plot(1:NumTestingSteps, [TradingShares_Step_TWAP_Test_Sell ...
    SimExecutedShares_Test_Sell], linewidth=1.5)
xlabel(strcat("Step number (", num2str(TradingIntervalSec), ...
    " sec intervals, ", num2str(NumTestingSteps), " steps total.)"))
ylabel("Executed Shares")
title("Shares Traded at Each Step (Testing, Sell)")
legend(["TWAP" "Agent"])
xlim([650 950])
ylim([0 2900])

subplot(3,1,3)
plot(1:NumTestingSteps, [InventoryShares_Step_TWAP_Test_Sell ...
    SimInventory_Test_Sell], linewidth=1.5)
xlabel(strcat("Step number (", num2str(TradingIntervalSec), ...
    " sec intervals, ", num2str(NumTestingSteps), " steps total.)"))
ylabel("Inventory Shares")
title("Inventory Shares (Testing, Sell)")
legend(["TWAP" "Agent"])
xlim([650 950])
```

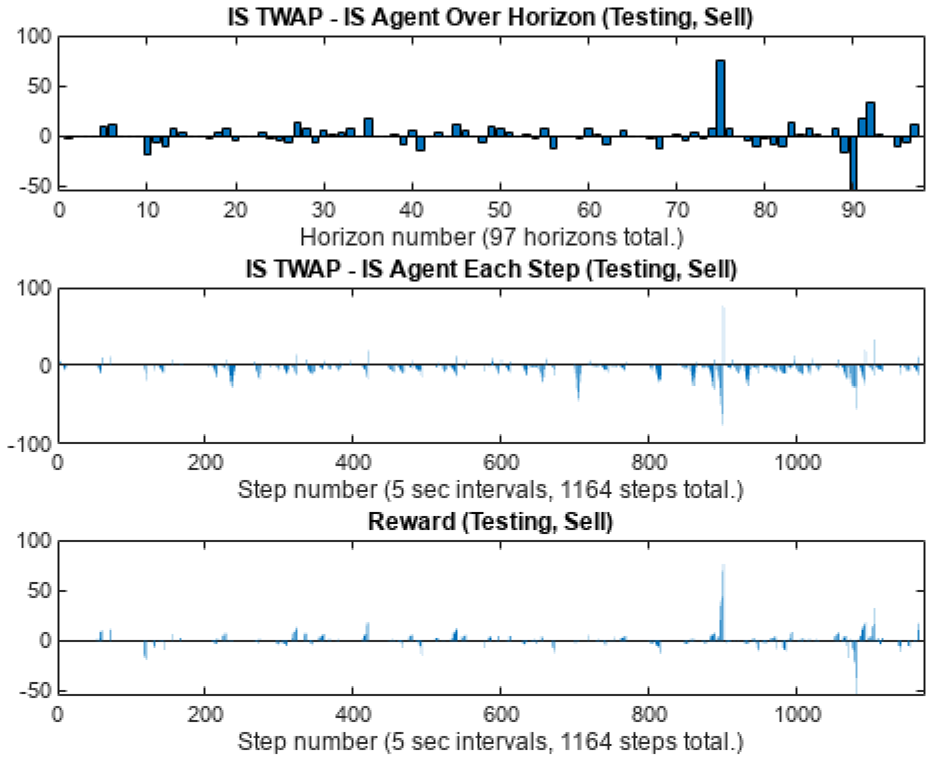


Just as in the training data, the agent (red) actively changes the number of executed shares for each step in the testing data, while the TWAP policy (blue) executes the same number of shares at each step.

Plot the differences in IS between TWAP and the agent and then compare them with the rewards.

```
figure
subplot(3,1,1)
bar(1:NumTestingHorizons, IS_TWAP_Horizon_Test_Sell - IS_Agent_Horizon_Test_Sell)
xlabel(strcat("Horizon number (", num2str(NumTestingHorizons), " horizons total.)"))
title("IS TWAP - IS Agent Over Horizon (Testing, Sell)")

subplot(3,1,2)
bar(1:NumTestingSteps, IS_TWAP_Step_Test_Sell - IS_Agent_Step_Test_Sell)
xlabel(strcat("Step number (", num2str(TradingIntervalSec), ...
    " sec intervals, ", num2str(NumTestingSteps), " steps total.)"))
title("IS TWAP - IS Agent Each Step (Testing, Sell)")
xlim([0 NumTestingSteps+9])
```

```
subplot(3,1,3)
bar(1:NumTestingSteps, SimReward_Test_Sell)
xlabel(strcat("Step number (", num2str(TradingIntervalSec), ...
    " sec intervals, ", num2str(NumTestingSteps), " steps total.)"))
title("Reward (Testing, Sell)")
xlim([0 NumTestingSteps+9])
```



Display Summary of Training and Testing Results for Sell Trades

You can summarize the results of the sell trade agent's training and testing.

Display total Implementation Shortfall (IS) outperformance of the agent over TWAP for training data.

```
Total_Outperformance_Train_Sell = sum(IS_TWAP_Horizon_Train_Sell - IS_Agent_Horizon_Train_Sell);
table(Total_Outperformance_Train_Sell)
```

```
ans=table
    Total_Outperformance_Train_Sell
    _______________________________

                164.47
```

Display total-gain-to-loss ratio (TGLR) and gain-to-loss ratio (GLR) for the training data.

```
[TGLR_Train_Sell, GLR_Train_Sell] = ...
    ISTGLR(IS_TWAP_Horizon_Train_Sell - IS_Agent_Horizon_Train_Sell);

table(TGLR_Train_Sell, GLR_Train_Sell)
```

```
ans=1×2 table
    TGLR_Train_Sell    GLR_Train_Sell
    _______________    ______________

        1.2327             1.2532
```

Display the total IS outperformance of the agent over TWAP for the testing data.

```
Total_Outperformance_Test_Sell = sum(IS_TWAP_Horizon_Test_Sell - IS_Agent_Horizon_Test_Sell);
table(Total_Outperformance_Test_Sell)
```

```
ans=table
    Total_Outperformance_Test_Sell
    ______________________________

                104.12
```

Display TGLR and GLR for the testing data.

```
[TGLR_Test_Sell, GLR_Test_Sell] = ...
    ISTGLR(IS_TWAP_Horizon_Test_Sell - IS_Agent_Horizon_Test_Sell);
table(TGLR_Test_Sell, GLR_Test_Sell)
```

```
ans=1×2 table
    TGLR_Test_Sell    GLR_Test_Sell
    ______________    _____________

        1.389            1.1906
```

Optimal Execution for Buy Trades

You can use a shortened version of the previous procedure (Optimal Execution for Sell Trades on page 4-391) to support the workflow for designing a DQN agent for optimal execution of buy trades. To repeat the same workflow for buy trades, first set the `BuyTrade` logical flag to `true` and define the total number of shares to buy over the horizon.

```
BuyTrade = true;
TotalTradingShares_Buy = 2551;
```

Execute TWAP trades and compute the Implementation Shortfall (IS).

```
[IS_TWAP_Horizon_Train_Buy,IS_TWAP_Step_Train_Buy, ...
    InventoryShares_Step_TWAP_Train_Buy,TradingShares_Step_TWAP_Train_Buy] = ...
    executeTWAPTrades(TrainingLOB,NumLevels,TotalTradingShares_Buy, ...
    NumIntervalsPerHorizon,NumTrainingHorizons,BuyTrade);
```

Set up the RL training environment for buy trades.

```
ObservationDimension = [1 4];
ObservationInfo_Train_Buy = rlNumericSpec(ObservationDimension);
ObservationInfo_Train_Buy.Name = 'Observation Data';
ObservationInfo_Train_Buy.Description = ['RemainingSharesToTrade, ...' ...
    'RemainingIntervals, IS_Agent, PriceDivergence'];
ObservationInfo_Train_Buy.LowerLimit = -inf;
ObservationInfo_Train_Buy.UpperLimit = inf;
```

```
EnvConstants_Train_Buy.SampledBook = TrainingLOB;
EnvConstants_Train_Buy.BuyTrade = BuyTrade;
EnvConstants_Train_Buy.TotalTradingShares = TotalTradingShares_Buy;
EnvConstants_Train_Buy.NumIntervalsPerHorizon = NumIntervalsPerHorizon;
EnvConstants_Train_Buy.NumHorizons = NumTrainingHorizons;
EnvConstants_Train_Buy.NumSteps = NumTrainingSteps;
EnvConstants_Train_Buy.NumLevels = NumLevels;
EnvConstants_Train_Buy.IS_TWAP = IS_TWAP_Step_Train_Buy;
ResetFunction_Buy = @() RL_OptimalExecution_LOB_ResetFcn(EnvConstants_Train_Buy);

NumActions = 39;
LowerActionRange = linspace(0,TradingShares_Step_TWAP_Train_Buy(1), round(NumActions/2));
ActionRange = round([LowerActionRange(1:end-1) ...
    linspace(TradingShares_Step_TWAP_Train_Buy(1), ...
    TradingShares_Step_TWAP_Train_Buy(1)*2,round(NumActions/2))]);
NumActions = length(ActionRange);

ActionInfo_Buy = rlFiniteSetSpec(ActionRange);
ActionInfo_Buy.Name = 'Trading Action (Number of Shares)';
ActionInfo_Buy.Description = 'Number of Shares to be Traded at Each Time Step';

StepFunction_Train_Buy = @(Action,LoggedSignals) ...
    RL_OptimalExecution_LOB_StepFcn(Action,LoggedSignals,EnvConstants_Train_Buy);

RL_OptimalExecution_Training_Environment_Buy = rlFunctionEnv( ...
    ObservationInfo_Train_Buy,ActionInfo_Buy,StepFunction_Train_Buy,ResetFunction_Buy)
```

```
Reset!
Reset!

RL_OptimalExecution_Training_Environment_Buy = 
  rlFunctionEnv with properties:

           StepFcn: @(Action,LoggedSignals)RL_OptimalExecution_LOB_StepFcn(Action,LoggedSignals,En
          ResetFcn: @()RL_OptimalExecution_LOB_ResetFcn(EnvConstants_Train_Buy)
    LoggedSignals: [1×1 struct]
```

Use Reinforcement Learning Designer (Reinforcement Learning Toolbox) to create and train the agent for the buy trades. You use the same procedure (Optimal Execution for Sell Trades on page 4-391) as with the sell trades, except that the imported training environment is for buy trades (`RL_OptimalExecution_Training_Environment_Buy`). Once the training is complete, export the trained agent to the workspace. Since training can take a long time, this example uses a pretrained agent (`DQN_agent_Buy_Trained.mat`).

```
load DQN_agent_Buy_Trained.mat
```

Compute IS using the trained agent and training data for buy trades.

```
rng('default');
reset(RL_OptimalExecution_Training_Environment_Buy);
```

```
Reset!
```

```
simOptions = rlSimulationOptions(MaxSteps=NumTrainingSteps);
Trained_Agent_Buy = DQN_agent_Buy_Trained;
experience_Buy = sim(RL_OptimalExecution_Training_Environment_Buy,Trained_Agent_Buy,simOptions);
```

```
Reset!
Last Horizon. Episode is Done!
HorizonIdx:

ans = 293
```

```
SimAction_Train_Buy = squeeze(experience_Buy.Action.TradingAction_NumberOfShares_.Data);
NumSimSteps = length(SimAction_Train_Buy);
SimInventory_Train_Buy = nan(NumSimSteps,1);
SimExecutedShares_Train_Buy = SimInventory_Train_Buy;
SimReward_Train_Buy = nan(NumSimSteps,1);
reset(RL_OptimalExecution_Training_Environment_Buy);
```

```
Reset!
```

```
for k=1:NumSimSteps
    [~,SimReward_Train_Buy(k),~,LoggedSignals] = ...
        step(RL_OptimalExecution_Training_Environment_Buy,SimAction_Train_Buy(k));
    SimInventory_Train_Buy(k) = LoggedSignals.CurrentInventoryShares;
    SimExecutedShares_Train_Buy(k) = sum(LoggedSignals.ExecutedShares);
end
```

```
Last Horizon. Episode is Done!
HorizonIdx:

ans = 293
```

```
IS_Agent_Step_Train_Buy = LoggedSignals.IS_Agent;
IS_Agent_Horizon_Train_Buy = IS_Agent_Step_Train_Buy( ....
    NumIntervalsPerHorizon:NumIntervalsPerHorizon:NumTrainingSteps);
```

Execute the TWAP trades and compute the IS for the testing data.

```
[IS_TWAP_Horizon_Test_Buy,IS_TWAP_Step_Test_Buy, ...
    InventoryShares_Step_TWAP_Test_Buy,TradingShares_Step_TWAP_Test_Buy] = ...
    executeTWAPTrades(TestingLOB,NumLevels,TotalTradingShares_Buy,...
    NumIntervalsPerHorizon,NumTestingHorizons,BuyTrade);
```

Set up the RL testing environment for buy trades.

```
EnvConstants_Test_Buy.SampledBook = TestingLOB;
EnvConstants_Test_Buy.BuyTrade = BuyTrade;
EnvConstants_Test_Buy.TotalTradingShares = TotalTradingShares_Buy;
EnvConstants_Test_Buy.NumIntervalsPerHorizon = NumIntervalsPerHorizon;
EnvConstants_Test_Buy.NumHorizons = NumTestingHorizons;
EnvConstants_Test_Buy.NumSteps = NumTestingSteps;
EnvConstants_Test_Buy.NumLevels = NumLevels;
EnvConstants_Test_Buy.IS_TWAP = IS_TWAP_Step_Test_Buy;
ResetFunction_Test_Buy = @() RL_OptimalExecution_LOB_ResetFcn(EnvConstants_Test_Buy);
StepFunction_Test_Buy = @(Action,LoggedSignals) ...
    RL_OptimalExecution_LOB_StepFcn(Action,LoggedSignals,EnvConstants_Test_Buy);
RL_OptimalExecution_Testing_Environment_Buy = ...
    rlFunctionEnv(ObservationInfo_Train_Buy,ActionInfo_Buy, ...
    StepFunction_Test_Buy,ResetFunction_Test_Buy)
```

```
Reset!
Reset!

RL_OptimalExecution_Testing_Environment_Buy = 
  rlFunctionEnv with properties:
          StepFcn: @(Action,LoggedSignals)RL_OptimalExecution_LOB_StepFcn(Action,LoggedSignals,E
         ResetFcn: @()RL_OptimalExecution_LOB_ResetFcn(EnvConstants_Test_Buy)
    LoggedSignals: [1×1 struct]
```

Compute IS using the trained agent and testing data for buy trades.

```
simOptions = rlSimulationOptions(MaxSteps=NumTestingSteps);
experience_Buy = sim(RL_OptimalExecution_Testing_Environment_Buy,Trained_Agent_Buy,simOptions);
```

```
Reset!
Last Horizon. Episode is Done!
HorizonIdx:

ans = 97
```

```
SimAction_Test_Buy = squeeze(experience_Buy.Action.TradingAction_NumberOfShares_.Data);
NumSimSteps = length(SimAction_Test_Buy);
SimInventory_Test_Buy = nan(NumSimSteps,1);
SimExecutedShares_Test_Buy = SimInventory_Test_Buy;
SimReward_Test_Buy = nan(NumSimSteps,1);
reset(RL_OptimalExecution_Testing_Environment_Buy);
```

```
Reset!
```

```
for k=1:NumSimSteps
    [NextObs1,SimReward_Test_Buy(k),IsDone1,LoggedSignals] = step( ...
        RL_OptimalExecution_Testing_Environment_Buy,SimAction_Test_Buy(k));
    SimInventory_Test_Buy(k) = LoggedSignals.CurrentInventoryShares;
    SimExecutedShares_Test_Buy(k) = sum(LoggedSignals.ExecutedShares);
end
```

```
Last Horizon. Episode is Done!
HorizonIdx:

ans = 97
```

```
IS_Agent_Step_Test_Buy = LoggedSignals.IS_Agent;
IS_Agent_Horizon_Test_Buy = IS_Agent_Step_Test_Buy( ...
    NumIntervalsPerHorizon:NumIntervalsPerHorizon:NumTestingSteps);
```

Display the total Implementation Shortfall (IS) outperformance of the agent over TWAP for training data.

```
Total_Outperformance_Train_Buy = sum(IS_TWAP_Horizon_Train_Buy - IS_Agent_Horizon_Train_Buy);
table(Total_Outperformance_Train_Buy)
```

```
ans=table
    Total_Outperformance_Train_Buy
    ______________________________

                326.25
```

Display total-gain-to-loss ratio (TGLR) and gain-to-loss ratio (GLR) for the training data.

```
[TGLR_Train_Buy, GLR_Train_Buy] = ...
    ISTGLR(IS_TWAP_Horizon_Train_Buy - IS_Agent_Horizon_Train_Buy);
table(TGLR_Train_Buy, GLR_Train_Buy)
```

```
ans=1×2 table
    TGLR_Train_Buy    GLR_Train_Buy
    ______________    _____________

        1.4146           1.2587
```

Display total IS outperformance of the agent over TWAP for the testing data.

```
Total_Outperformance_Test_Buy = sum(IS_TWAP_Horizon_Test_Buy - IS_Agent_Horizon_Test_Buy);
table(Total_Outperformance_Test_Buy)
```

```
ans=table
    Total_Outperformance_Test_Buy
    _____________________________

               196.57
```

Display TGLR and GLR for the testing data.

```
[TGLR_Test_Buy, GLR_Test_Buy] = ...
    ISTGLR(IS_TWAP_Horizon_Test_Buy - IS_Agent_Horizon_Test_Buy);
table(TGLR_Test_Buy, GLR_Test_Buy)
```

```
ans=1×2 table
    TGLR_Test_Buy    GLR_Test_Buy
    _____________    ____________

        2.018           1.4641
```

References

[1] Bertsimas, Dimitris, and Andrew W. Lo. "Optimal Control of Execution Costs." *Journal of Financial Markets* 1, no. 1 (1998): 1-50.

[2] Almgren, Robert, and Neil Chriss. "Optimal Execution of Portfolio Transactions." *Journal of Risk* 3, no. 2 (2000): 5-40.

[3] Nevmyvaka, Yuriy, Yi Feng, and Michael Kearns. "Reinforcement Learning for Optimized Trade Execution." In *Proceedings of the 23rd International Conference on Machine Learning*, pp. 673-680. 2006.

[4] Ning B., F. Lin, and S. Jaimungal. "Double Deep Q-Learning for Optimal Execution." Preprint, submitted June 8, 2020. Available at https://arxiv.org/abs/1812.06600.

[5] Lin S. and P. A. Beling. "Optimal Liquidation with Deep Reinforcement Learning." *33rd Conference on Neural Information Processing Systems (NeurIPS 2019) Deep Reinforcement Learning Workshop*. Vancouver, Canada, 2019.

[6] Lin S. and P. A. Beling. "An End-to-End Optimal Trade Execution Framework Based on Proximal Policy Optimization." *Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI 2020)* Special Track on AI in FinTech. 4548-4554. 2020.

[7] LOBSTER Limit Order Book Data. Berlin: frischedaten UG (haftungsbeschränkt). Available at https://lobsterdata.com/.

Local Functions

```
function [IS_TWAP_Horizon,IS_TWAP_Step,InventorySharesStepTWAP,TradingSharesStepTWAP] = ...
    executeTWAPTrades(InputLOB,NumLevels,TotalTradingShares,NumIntervalsPerHorizon,NumHorizons,Bu
% executeTWAPTrades Function for Executing Trades for TWAP Baseline
% This function executes buy or sell trades for the TWAP baseline and
% computes the corresponding implementation shortfalls, inventory shares
% and trading shares at each step.

NumSteps = NumHorizons*NumIntervalsPerHorizon;
IS_TWAP_Horizon = nan(NumHorizons,1);
IS_TWAP_Step = nan(NumSteps,1);
TradingSharesStepTWAP = round(TotalTradingShares./NumIntervalsPerHorizon).*ones(NumSteps,1);

if BuyTrade
    InventorySharesStepTWAP = zeros(NumSteps,1);
else
    InventorySharesStepTWAP = zeros(NumSteps,1) + TotalTradingShares;
end

for HorizonIdx = 1:NumHorizons
    HorizonExecutedPrices = zeros(NumLevels*NumIntervalsPerHorizon,1);
    HorizonExecutedShares = HorizonExecutedPrices;
    for IntervalIdx = 1:NumIntervalsPerHorizon
        CurrentBook =  InputLOB(IntervalIdx + NumIntervalsPerHorizon*(HorizonIdx-1),:);

        % Execute trade
        if IntervalIdx==NumIntervalsPerHorizon
            if BuyTrade
                % Enforce target ending inventory constraint for buy trade (buy missing shares)
                CurrentTradingShares = max(TotalTradingShares - ...
                    InventorySharesStepTWAP(IntervalIdx + NumIntervalsPerHorizon.*(HorizonIdx-1)
            else
                % Enforce zero ending inventory constraint for sell trade (sell remaining shares
                CurrentTradingShares = max(InventorySharesStepTWAP(IntervalIdx + ...
                    NumIntervalsPerHorizon.*(HorizonIdx-1)), 0);
            end
        else
            CurrentTradingShares = TradingSharesStepTWAP(IntervalIdx + ...
                NumIntervalsPerHorizon.*(HorizonIdx-1));
        end

        [ExecutedShares,ExecutedPrices,InitialPrice,TradedLevels] = ...
            tradeLOB(CurrentBook,CurrentTradingShares,BuyTrade);
        if IntervalIdx==1
            ArrivalPrice = InitialPrice;
        end
        HorizonExecutedShares((1:TradedLevels) + (IntervalIdx-1)*NumLevels) = ExecutedShares;
        HorizonExecutedPrices((1:TradedLevels) + (IntervalIdx-1)*NumLevels) = ExecutedPrices;
        if BuyTrade
            InventorySharesStepTWAP((IntervalIdx:NumIntervalsPerHorizon) + ...
                NumIntervalsPerHorizon.*(HorizonIdx-1)) = ...
                InventorySharesStepTWAP(IntervalIdx + ...
                NumIntervalsPerHorizon.*(HorizonIdx-1)) + sum(ExecutedShares);
        else
            InventorySharesStepTWAP((IntervalIdx:NumIntervalsPerHorizon) + ...
                NumIntervalsPerHorizon.*(HorizonIdx-1)) = ...
                InventorySharesStepTWAP(IntervalIdx + ...
                NumIntervalsPerHorizon.*(HorizonIdx-1)) - sum(ExecutedShares);
        end

        % Compute Implementation Shortfall for each Step
        IS_TWAP_Step(IntervalIdx + NumIntervalsPerHorizon.*(HorizonIdx-1)) = ...
            endingInventoryIS(ArrivalPrice,HorizonExecutedPrices,HorizonExecutedShares,BuyTrade)
    end

    % Compute Implementation Shortfall for each Horizon
    IS_TWAP_Horizon(HorizonIdx) = endingInventoryIS(ArrivalPrice, ...
        HorizonExecutedPrices,HorizonExecutedShares,BuyTrade);
end

end


function [InitialObservation,LoggedSignals] = RL_OptimalExecution_LOB_ResetFcn(EnvConstants)
% RL_OptimalExecution_LOB_ResetFcn is Reset Function for RL Agent
% Define Reset Function
%
% Inputs:
%
% EnvConstants: Structure with the following fields:
% - SampledBook: Limit order book sampled at time intervals
% - BuyTrade: Scalar logical indicating the trading direction
%               true: Buy
%               false: Sell
% - TotalTradingShares: Number of shares to trade over the horizon (scalar)
% - NumIntervalsPerHorizon: Number of intervals per horizon (scalar)
% - NumHorizons: Number of horizons in the data (scalar)
% - NumSteps: Number of steps in the data (scalar)
% - NumLevels: Number of levels in the limit order book (scalar)
% - IS_TWAP: Implementation shortfall for TWAP baseline (NumSteps-by-1 vector)
%
% Outputs:
%
% InitialObservation:
% - Remaining shares to be traded by agent
% - Remaining intervals (steps) in the current trading horizon
% - Implementation shortfall of agent
% - Price divergence times 100
%
% LoggedSignals: Structure with following fields:
% - IntervalIdx: Current time interval index (initial value: 0)
% - HorizonIdx: Current time horizon index (initial value: 0)
% - CurrentInventoryShares: Current number of shares in inventory
%    (Initial value: 0 for Buy trade)
%    (Initial value: TotalTradingShares for Sell trade)
% - ArrivalPrice: Arrival price at the beginning of horizon (initial value: 0)
% - ExecutedShares: Number of executed shares at each traded level (initial value: 0)
% - ExecutedPrices: Prices of executed shares at each traded level (initial value: 0)
% - HorizonExecutedShares: Executed shares at each level over the horizon
%     (Initial value: (NumLevels * NumIntervalsPerHorizon)-by-1 vector of zeros)
% - HorizonExecutedPrices: Prices of Executed shares at each level over the horizon
%     (Initial value: (NumLevels * NumIntervalsPerHorizon)-by-1 vector of zeros)
% - IS_Agent: Implementation shortfall history of agent
%     (Initial value: NumIntervals-by-1 vector of zeros)
% - Reward: Reward history of agent
%     (Initial value: NumIntervals-by-1 vector of zeros)
% - NumIntervalsPerHorizon: Number of intervals per horizon (fixed value: NumIntervalsPerHorizon)
% - NumHorizons: Number of horizons (fixed value: NumHorizon)
% - NumLevels: Number of levels in the limit order book (fixed value: NumLevels)

IntervalIdx = 0;
HorizonIdx = 0;

LoggedSignals.IntervalIdx = IntervalIdx;
LoggedSignals.HorizonIdx = HorizonIdx;

InitialObservation = ...
    [EnvConstants.TotalTradingShares EnvConstants.NumIntervalsPerHorizon 0 0];

if EnvConstants.BuyTrade
    LoggedSignals.CurrentInventoryShares = 0;
else
    LoggedSignals.CurrentInventoryShares = EnvConstants.TotalTradingShares;
end

LoggedSignals.ArrivalPrice = 0;
LoggedSignals.ExecutedShares = 0;
LoggedSignals.ExecutedPrices = 0;
LoggedSignals.HorizonExecutedShares = ...
    zeros(EnvConstants.NumLevels*EnvConstants.NumIntervalsPerHorizon,1);
LoggedSignals.HorizonExecutedPrices = LoggedSignals.HorizonExecutedShares;
LoggedSignals.IS_Agent = zeros(EnvConstants.NumSteps,1);
LoggedSignals.Reward = LoggedSignals.IS_Agent;
LoggedSignals.NumIntervalsPerHorizon = EnvConstants.NumIntervalsPerHorizon;
LoggedSignals.NumHorizons = EnvConstants.NumHorizons;
LoggedSignals.NumLevels = EnvConstants.NumLevels;

disp("Reset!")
end

function [Observation,Reward,IsDone,LoggedSignals] = ...
    RL_OptimalExecution_LOB_StepFcn(Action,LoggedSignals,EnvConstants)
% RL_OptimalExecution_LOB_StepFcn function is Step Function for RL Environment
%   Given current Action and LoggedSignals, update Observation, Reward, and
%   LoggedSignals, and indicate whether the episode is complete.
%
% Inputs:
%
% Action: Number of shares to be traded at the current step (scalar)
%
% LoggedSignals: Structure with following fields
% - IntervalIdx: Current time interval index
% - HorizonIdx: Current time horizon index
% - CurrentInventoryShares: Current number of shares in inventory
% - ArrivalPrice: Arrival price at the beginning of horizon
% - ExecutedShares: Number of executed shares at each traded level
% - ExecutedPrices: Prices of executed shares at each traded level
% - HorizonExecutedShares: Executed shares at each level over the horizon
%     ((NumLevels * NumIntervalsPerHorizon)-by-1 vector)
% - HorizonExecutedPrices: Prices of Executed shares at each level over the horizon
%     ((NumLevels * NumIntervalsPerHorizon)-by-1 vector)
% - IS_Agent: Implementation shortfall history of agent
%     (NumIntervals-by-1 vector)
% - Reward: Reward history of agent
%     (NumIntervals-by-1 vector)
% - NumIntervalsPerHorizon: Number of intervals per horizon (fixed value: NumIntervalsPerHorizon)
% - NumHorizons: Number of horizons (fixed value: NumHorizon)
% - NumLevels: Number of levels in the limit order book (fixed value: NumLevels)
%
% EnvConstants: Structure with the following fields
% - SampledBook: Limit order book sampled at time intervals
% - BuyTrade: Scalar logical indicating the trading direction
%               true: Buy
%               false: Sell
% - TotalTradingShares: Number of shares to trade over the horizon (scalar)
% - NumIntervalsPerHorizon: Number of intervals per horizon (scalar)
% - NumHorizons: Number of horizons in the data (scalar)
% - NumSteps: Number of steps in the data (scalar)
% - NumLevels: Number of levels in the limit order book (scalar)
% - IS_TWAP: Implementation shortfall for TWAP baseline (NumSteps-by-1 vector)
%
% Outputs:
%
% Observation:
% - Remaining shares to be traded by agent
% - Remaining intervals (steps) in the current trading horizon
% - Implementation shortfall of agent
% - Price divergence times 100
%
% Reward: Reward for current step
%
% IsDone: Logical indicating whether the current episode is complete
%
% LoggedSignals: Updated LoggedSignals

% Update LoggedSignals indices
if (LoggedSignals.IntervalIdx == 0) && (LoggedSignals.HorizonIdx == 0)
    LoggedSignals.IntervalIdx = 1;
    LoggedSignals.HorizonIdx = 1;
elseif (LoggedSignals.IntervalIdx >= LoggedSignals.NumIntervalsPerHorizon)
    LoggedSignals.IntervalIdx = 1;
    LoggedSignals.HorizonIdx = LoggedSignals.HorizonIdx + 1;
    LoggedSignals.HorizonExecutedShares = ...
        zeros(EnvConstants.NumLevels*EnvConstants.NumIntervalsPerHorizon,1);
    LoggedSignals.HorizonExecutedPrices = LoggedSignals.HorizonExecutedShares;
    if EnvConstants.BuyTrade
        LoggedSignals.CurrentInventoryShares = 0;
    else
        LoggedSignals.CurrentInventoryShares = EnvConstants.TotalTradingShares;
    end
else
    LoggedSignals.IntervalIdx = LoggedSignals.IntervalIdx + 1;
```

```
end

StepIdx = LoggedSignals.IntervalIdx + ...
    LoggedSignals.NumIntervalsPerHorizon.*(LoggedSignals.HorizonIdx-1);
RemainingIntervals = LoggedSignals.NumIntervalsPerHorizon - LoggedSignals.IntervalIdx + 1;
IS_TWAP = EnvConstants.IS_TWAP(StepIdx);


if EnvConstants.BuyTrade
    AvgPrice = mean(EnvConstants.SampledBook(StepIdx,[1 5]),2);
    PriceDivergence = LoggedSignals.ArrivalPrice - AvgPrice;
else
    AvgPrice = mean(EnvConstants.SampledBook(StepIdx,[3 7]),2);
    PriceDivergence = AvgPrice - LoggedSignals.ArrivalPrice;
end

% Enforce ending inventory constraint
if LoggedSignals.IntervalIdx >= LoggedSignals.NumIntervalsPerHorizon
    if EnvConstants.BuyTrade
        TradingShares = max(EnvConstants.TotalTradingShares - ...
            LoggedSignals.CurrentInventoryShares, 0);
    else
        TradingShares = LoggedSignals.CurrentInventoryShares;
    end
else
    if EnvConstants.BuyTrade
        if PriceDivergence > 0
            AgentRemainingSharestoTrade = max(EnvConstants.TotalTradingShares - ...
                LoggedSignals.CurrentInventoryShares,0);
            TradingShares = round(AgentRemainingSharestoTrade./RemainingIntervals./2);
            TradingShares = min(TradingShares,Action);
        else
            TradingShares = min(EnvConstants.TotalTradingShares - ...
                LoggedSignals.CurrentInventoryShares, Action);
        end
    else
        if PriceDivergence > 0
            AgentRemainingSharestoTrade = max(LoggedSignals.CurrentInventoryShares, 0);
            TradingShares = round(AgentRemainingSharestoTrade./RemainingIntervals./2);
            TradingShares = min(TradingShares,Action);
        else
            TradingShares = min(LoggedSignals.CurrentInventoryShares, Action);
        end
    end
end

% Update IsDone:
% Episode is complete (IsDone is true) when time step reaches the end of
% the trading data.
if (LoggedSignals.HorizonIdx >= LoggedSignals.NumHorizons) && ...
        (LoggedSignals.IntervalIdx >= LoggedSignals.NumIntervalsPerHorizon)
    IsDone = true;
    disp("Last Horizon. Episode is Done!");
    disp("HorizonIdx:");
    LoggedSignals.HorizonIdx
elseif (LoggedSignals.IntervalIdx >= LoggedSignals.NumIntervalsPerHorizon)
    IsDone = false;
    % disp("HorizonIdx:");
```

```
        % LoggedSignals.HorizonIdx
    else
        IsDone = false;
    end

    % Execute trade
    CurrentLOB = EnvConstants.SampledBook(LoggedSignals.IntervalIdx + ...
        LoggedSignals.NumIntervalsPerHorizon*(LoggedSignals.HorizonIdx-1),:);

    [ExecutedShares,ExecutedPrices,InitialPrice,TradedLevels] = ...
        tradeLOB(CurrentLOB,TradingShares,EnvConstants.BuyTrade);

    % Update Reward
    if LoggedSignals.IntervalIdx==1
        LoggedSignals.ArrivalPrice = InitialPrice;
    end

    LoggedSignals.HorizonExecutedShares((1:TradedLevels) + ...
        (LoggedSignals.IntervalIdx-1)*LoggedSignals.NumLevels) = ExecutedShares;
    LoggedSignals.HorizonExecutedPrices((1:TradedLevels) + ...
        (LoggedSignals.IntervalIdx-1)*LoggedSignals.NumLevels) = ExecutedPrices;


    IS_Agent = endingInventoryIS(LoggedSignals.ArrivalPrice,LoggedSignals.HorizonExecutedPrices, ...
        LoggedSignals.HorizonExecutedShares,EnvConstants.BuyTrade);

    if EnvConstants.BuyTrade
        AgentRemainingSharestoTrade = max(EnvConstants.TotalTradingShares - ...
            LoggedSignals.CurrentInventoryShares - sum(ExecutedShares),0);
    else
        AgentRemainingSharestoTrade = max(LoggedSignals.CurrentInventoryShares - ...
            sum(ExecutedShares), 0);
    end

    [PenaltyAgentExecutedShares,PenaltyAgentExecutedPrices] = tradeLOB(CurrentLOB, ...
        round(AgentRemainingSharestoTrade./RemainingIntervals),EnvConstants.BuyTrade);

    IS_Agent_Penalty = RemainingIntervals.*endingInventoryIS(LoggedSignals.ArrivalPrice, ...
        PenaltyAgentExecutedPrices,PenaltyAgentExecutedShares,EnvConstants.BuyTrade);

    TWAPRemainingSharestoTrade = (EnvConstants.NumIntervalsPerHorizon - LoggedSignals.IntervalIdx)*
        EnvConstants.TotalTradingShares/EnvConstants.NumIntervalsPerHorizon;

    [PenaltyTWAPExecutedShares,PenaltyTWAPExecutedPrices] = tradeLOB(CurrentLOB, ...
        round(TWAPRemainingSharestoTrade./RemainingIntervals),EnvConstants.BuyTrade);

    IS_TWAP_Penalty = RemainingIntervals.*endingInventoryIS(LoggedSignals.ArrivalPrice, ...
        PenaltyTWAPExecutedPrices,PenaltyTWAPExecutedShares,EnvConstants.BuyTrade);

    Reward = ((IS_TWAP + IS_TWAP_Penalty) - (IS_Agent + IS_Agent_Penalty)) .* ...
        LoggedSignals.IntervalIdx./ LoggedSignals.NumIntervalsPerHorizon;

    % No reward if trading target already met
    if EnvConstants.BuyTrade
        if LoggedSignals.CurrentInventoryShares >= EnvConstants.TotalTradingShares
            Reward = 0;
        end
    else
```

```
        if LoggedSignals.CurrentInventoryShares <= 0
            Reward = 0;
        end
    end

    % Update LoggedSignals
    LoggedSignals.ExecutedPrices = ExecutedPrices;
    LoggedSignals.ExecutedShares = ExecutedShares;

    if EnvConstants.BuyTrade
        LoggedSignals.CurrentInventoryShares = ...
            LoggedSignals.CurrentInventoryShares + sum(ExecutedShares);
    else
        LoggedSignals.CurrentInventoryShares = ...
            max(LoggedSignals.CurrentInventoryShares - sum(ExecutedShares), 0);
    end

    % Update LoggedSignals.IS_Agent
    LoggedSignals.IS_Agent(StepIdx) = IS_Agent;

    % Update LoggedSignals.Reward
    LoggedSignals.Reward(StepIdx) = Reward;

    % Update Observation
    if LoggedSignals.IntervalIdx == LoggedSignals.NumIntervalsPerHorizon
        Observation = ...
            [EnvConstants.TotalTradingShares RemainingIntervals-1 IS_Agent PriceDivergence*100];
    else
        Observation = ...
            [AgentRemainingSharestoTrade RemainingIntervals-1 IS_Agent PriceDivergence*100];
    end

    end


    function [ExecutedShares,ExecutedPrices,InitialPrice,TradedLevels] = ...
        tradeLOB(CurrentLOB,TradingShares,BuyTrade)
    % tradeLOB function executes market order trade based on current limit order book.
    %   This function computes the results of executing a market order trade
    %   for a specified number of shares and trade direction (buy/sell) based on
    %   the information available in the current snapshot of the limit order book.
    %
    %   Inputs:
    %
    %      CurrentLOB - 1-by-(NumLevels*4) vector of current limit order book information
    %
    %         e.g. For NumLevels==5, 1-by-20 vector with the following values:
    %     [AskPrice1 AskSize1 BidPrice1 BidSize1 ... AskPrice5 AskSize5 BidPrice5 BidSize5]
    %
    %         e.g. For NumLevels==10, 1-by-40 vector with the following values:
    %     [AskPrice1 AskSize1 BidPrice1 BidSize1 ... AskPrice10 AskSize10 BidPrice10 BidSize10]
    %
    %      TradingShares - Scalar number of shares to be traded
    %
    %      BuyTrade - Scalar logical indicating trading direction:
    %                     true: Buy
    %                     false: Sell
    %
```

```
%    Outputs:
%      ExecutedShares - Vector of number of executed shares at each traded level
%      ExecutedPrices - Vector of number of executed prices at each traded level
%      InitialPrice - Scalar numeric initial price
%      TradedLevels - Scalar numeric traded levels

LOBLength = length(CurrentLOB);

% Separate Bid and Ask data
AskPriceIdx = (1:4:LOBLength);
AskSizeIdx = AskPriceIdx + 1;
BidPriceIdx = AskSizeIdx + 1;
BidSizeIdx = BidPriceIdx + 1;

AskPrices = CurrentLOB(AskPriceIdx)';
AskSizes = CurrentLOB(AskSizeIdx)';
BidPrices = CurrentLOB(BidPriceIdx)';
BidSizes = CurrentLOB(BidSizeIdx)';

if BuyTrade
    LOBPrices = AskPrices;
    LOBSizes = AskSizes;
else
    LOBPrices = BidPrices;
    LOBSizes = BidSizes;
end

InitialPrice = LOBPrices(1);

% Execute trade
CumulativeLOBSizes = cumsum(LOBSizes);
TradedLevels = sum(CumulativeLOBSizes <= TradingShares);
if TradedLevels < 1
    TradedLevels = 1;
elseif CumulativeLOBSizes(TradedLevels) < TradingShares
    TradedLevels = TradedLevels + 1;
end

ExecutedPrices = LOBPrices(1:TradedLevels);
ExecutedShares = LOBSizes(1:TradedLevels);
ExecutedShares(end) = ExecutedShares(end) - (sum(ExecutedShares) - TradingShares);
end


function outIS = endingInventoryIS(ArrivalPrice,ExecutedPrices,TradedVolumes,BuyTrade)
% EndingInventoryIS function for Implementation Shortfall given ending inventory constraints
%   This function computes the Implementation Shortfall (IS) under the
%   assumption of ending inventory constraints (i.e. no opportunity cost).
%
%   For a Sell trade, the constraint is to have zero ending inventory
%   assuming successful liquidation of all shares.
%
%   For a Buy trade, the constraint is to have full ending inventory
%   assuming successful purchase of all shares.
%
%   Inputs:
%
%      ArrivalPrice - Scalar price immediately before trade execution
```

```
%
%      ExecutedPrices - Vector of executed prices
%
%      TradedVolumes - Vector of executed trade volumes
%
%      BuyTrade - Scalar logical indicating trading direction:
%                    true: Buy
%                    false: Sell
%
%    Outputs:
%      outIS - Implementation shortfall

outIS = ArrivalPrice.*sum(TradedVolumes) - sum(ExecutedPrices.*TradedVolumes);

if BuyTrade
    outIS = -outIS;
end

end


function [TGLR, GLR] = ISTGLR(ISDiffValues)
% ISTGLR function for Total-Gain-to-Loss Ratio for Implementation Shortfalls (IS)
%   This function computes the Total-Gain-to-Loss Ratio (TGLR) and the
%   Gain-to-Loss Ratio (GLR) for a vector of values (IS_Baseline - IS_Agent)
%   where:
%     - IS_Baseline: Implementation shortfalls for baseline (e.g. TWAP)
%     - IS_Agent: Implementation shortfalls for agent
%
%   Input:
%     - ISDiffValues: Vector of IS difference values (IS_Baseline - IS_Agent)
%
%   Outputs:
%     - TGLR: Total-Gain-to-Loss Ratio computed as the ratio of
%             Total Positive over Total Negative magnitudes of IS difference
%     - GLR:  Gain-to-Loss Ratio computed as the ratio of
%             Mean Positive over Mean Negative magnitudes of IS difference

ISDiffValues = ISDiffValues(:);

ISGains = ISDiffValues(ISDiffValues>0);
ISLosses = ISDiffValues(ISDiffValues<0);

MeanGains = abs(mean(ISGains));
MeanLosses = abs(mean(ISLosses));

TGLR = sum(ISGains)./abs(sum(ISLosses));
GLR = MeanGains./MeanLosses;

end
```

See Also

Related Examples

- “Hedge Options Using Reinforcement Learning Toolbox™” on page 10-40

- “Use Deep Learning to Approximate Barrier Option Prices with Heston Model” (Financial Instruments Toolbox)

CVaR Portfolio Optimization Tools

- “Compute Maximum Reward-to-Risk Ratio for CVaR Portfolio” on page 5-130

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-5) 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 supports mean-variance portfolio optimization (see Markowitz [46], [47] at “Portfolio Optimization” on page A-5). 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 implements what is known as conditional value-at-risk portfolio optimization (see Rockafellar and Uryasev [48], [49] at “Portfolio Optimization” on page A-5), 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 implements what is known as mean-absolute deviation portfolio optimization (see Konno and Yamazaki [50] at “Portfolio Optimization” on page A-5), 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 $\mu: X \rightarrow R$ 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 $y_1,...,y_S$ 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

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

where:

r_0 is the risk-free rate (scalar).

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

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

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

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

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

Risk Proxy

The proxy for portfolio risk is a function $\sigma\colon X \to R$ 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

$$Variance(x) = x^T C x$$

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

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-5).

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

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

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 $s = 1,...,S$. 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

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

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-5.

Mean Absolute-Deviation

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

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

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-5.

See Also

`PortfolioCVaR`

Related Examples

- “Creating the PortfolioCVaR Object” on page 5-22
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-50

More About

- “PortfolioCVaR Object” on page 5-17
- “Portfolio Set for Optimization Using PortfolioCVaR Object” on page 5-8
- “Default Portfolio Problem” on page 5-15
- “PortfolioCVaR Object Workflow” on page 5-16

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- CVaR Portfolio Optimization (4 min 56 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 and which are properties for the `PortfolioCVaR` object:

- Linear inequality constraints
- Linear equality constraints
- `'Simple'` Bound constraints
- `'Conditional'` Bond constraints
- Budget constraints
- Group constraints
- Group ratio constraints
- Average turnover constraints
- One-way turnover constraints
- Cardinality constraints

Linear Inequality Constraints

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

$$A_I x \leq b_I$$

where:

x is the portfolio (n vector).

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

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

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

`PortfolioCVaR` 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.

`PortfolioCVaR` 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.

'Simple' Bound Constraints

`'Simple'` 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.

`PortfolioCVaR` 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 5-15) has l_B = `0` with u_B set implicitly through a budget constraint.

'Conditional' Bound Constraints

`'Conditional'` Bound constraints, also called semicontinuous constraints, are mixed-integer linear constraints that confine portfolio weights to fall either above or below specific bounds *if* the asset is selected; otherwise, the value of the asset is zero. Use `setBounds` to specify bound constraints with a `'Conditional'` `BoundType`. To mathematically formulate this type of constraints, a binary variable v_i is needed. $v_i = 0$ indicates that asset i is not selected and v_i indicates that the asset was selected. Thus

$$l_i v_i \leq x_i \leq u_i v_i$$

where

x is the portfolio (n vector).

l is the conditional lower-bound constraint (n vector).

u is the conditional upper-bound constraint (n vector).

n is the number of assets in the universe.

`PortfolioCVaR` object properties to specify the bound constraint are:

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

The default is to ignore this constraint.

Budget Constraints

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

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

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.

`PortfolioCVaR` 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 5-15) 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 5-45).

Group Constraints

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

$$l_G \leq Gx \leq u_G$$

where:

x is the portfolio (n vector).

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

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

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

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

PortfolioCVaR 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

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

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.

`PortfolioCVaR` 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

$$\frac{1}{2}1^T\left|x - x_0\right| \le \tau$$

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.

`PortfolioCVaR` 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

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

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

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 `PortfolioMAD` object:

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

The default is to ignore this constraint.

Note The average turnover constraint (see “Average Turnover Constraints” on page 5-12) with τ is not a combination of the one-way turnover constraints with $\tau = \tau_B = \tau_S$.

Cardinality Constraints

Cardinality constraint limits the number of assets in the optimal allocation for an `PortfolioCVaR` object. Use `setMinMaxNumAssets` to specify the `'MinNumAssets'` and `'MaxNumAssets'` constraints. To mathematically formulate this type of constraints, a binary variable v_i is needed. $v_i = 0$ indicates that asset i is not selected and $v_i = 1$ indicates that the asset was selected. Thus

$$MinNumAssets \leq \sum_{i = 1}^{NumAssets} v_i \leq MaxNumAssets$$

The default is to ignore this constraint.

See Also

`PortfolioCVaR`

Related Examples

- “Creating the PortfolioCVaR Object” on page 5-22
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-50
- “Hedging Using CVaR Portfolio Optimization” on page 5-118
- “Compute Maximum Reward-to-Risk Ratio for CVaR Portfolio” on page 5-130

More About

- “PortfolioCVaR Object” on page 5-17
- “Default Portfolio Problem” on page 5-15
- “PortfolioCVaR Object Workflow” on page 5-16

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- CVaR Portfolio Optimization (4 min 56 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 5-22
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-50
- “Hedging Using CVaR Portfolio Optimization” on page 5-118
- “Compute Maximum Reward-to-Risk Ratio for CVaR Portfolio” on page 5-130

More About

- “PortfolioCVaR Object” on page 5-17
- “Portfolio Set for Optimization Using PortfolioCVaR Object” on page 5-8
- “PortfolioCVaR Object Workflow” on page 5-16

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- CVaR Portfolio Optimization (4 min 56 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 5-22.
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 5-36.
3. Specify the CVaR Portfolio Constraints.

   Define the constraints for portfolio assets such as linear equality and inequality, bound, budget, group, group ratio, turnover constraints, `'Conditional'` `BoundType`, and `MinNumAssets`, `MaxNumAssets` constraints. For more information, see “Working with CVaR Portfolio Constraints Using Defaults” on page 5-50 and “Working with 'Conditional' BoundType, MinNumAssets, and MaxNumAssets Constraints Using PortfolioCVaR Objects” on page 5-69.
4. Validate the CVaR Portfolio.

   Identify errors for the portfolio specification. For more information, see “Validate the CVaR Portfolio Problem” on page 5-78.
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 5-82 and “Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-102.
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 5-110.

See Also

Related Examples

- “Hedging Using CVaR Portfolio Optimization” on page 5-118
- “Compute Maximum Reward-to-Risk Ratio for CVaR Portfolio” on page 5-130

More About

- “Portfolio Optimization Theory” on page 5-3

External Websites

- Getting Started with Portfolio Optimization (4 min 13 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 `PortfolioCVaR` 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 `PortfolioCVaR` to set the properties for the `PortfolioCVaR` object.

`PortfolioCVaR` 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 5-22.

Setting and Getting Properties

You can set properties of a PortfolioCVaR object using either the PortfolioCVaR object 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 object 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', 1, 'ProbabilityLevel', 0.95);
```

In addition to the PortfolioCVaR object, 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 object and set functions have several useful features:

- The PortfolioCVaR object and set functions try to determine the dimensions of your problem with either explicit or implicit inputs.
- The PortfolioCVaR object and set functions try to resolve ambiguities with default choices.
- The PortfolioCVaR object 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. An efficient portfolio are the portfolios that satisfy the criteria of minimum risk for a given level of return and maximum return for a given level of risk. 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)
```

```
disp(p)
  3×2 PortfolioCVaR array 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
    MinNumAssets
    MaxNumAssets
    BoundType
```

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 PortfolioCVaR 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 5-22
- "Working with CVaR Portfolio Constraints Using Defaults" on page 5-50
- "Hedging Using CVaR Portfolio Optimization" on page 5-118
- "Compute Maximum Reward-to-Risk Ratio for CVaR Portfolio" on page 5-130

More About

- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-16

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- CVaR Portfolio Optimization (4 min 56 sec)

Creating the PortfolioCVaR Object

In this section...

To create a fully specified CVaR portfolio optimization problem, instantiate the PortfolioCVaR object using PortfolioCVaR. For information on the workflow when using PortfolioCVaR objects, see "PortfolioCVaR Object Workflow" on page 5-16.

Syntax

Use PortfolioCVaR to create an instance of an object of the PortfolioCVaR class. You can use PortfolioCVaR object 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 object also accepts collections of argument name-value pair arguments for properties and their values. The PortfolioCVaR object 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 PortfolioCVaR 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 5-26). The PortfolioCVaR object 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: []
        MinNumAssets: []
        MaxNumAssets: []
           BoundType: []
```

The approaches listed provide a way to set up a portfolio optimization problem with the `PortfolioCVaR` object. 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 object 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:

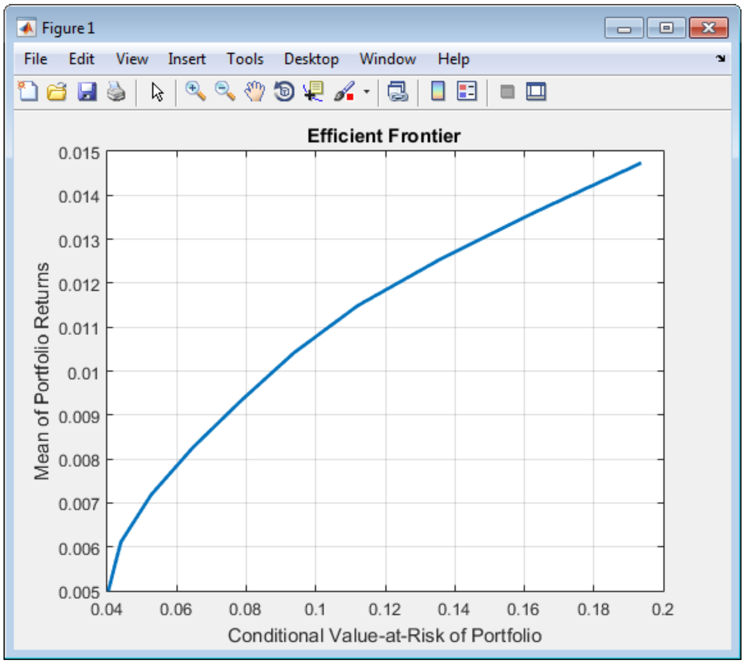
```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioCVaR('Scenarios', AssetScenarios, ...
'LowerBound', 0, 'LowerBudget', 1, 'UpperBudget', 1, ...
'ProbabilityLevel', 0.95)
```

```
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: [4×1 double]
          UpperBound: []
         LowerBudget: 1
         UpperBudget: 1
         GroupMatrix: []
          LowerGroup: []
          UpperGroup: []
              GroupA: []
              GroupB: []
          LowerRatio: []
          UpperRatio: []
        MinNumAssets: []
        MaxNumAssets: []
           BoundType: []
```

The LowerBound property value undergoes scalar expansion since AssetScenarios provides the dimensions of the problem.

You can use dot notation with the function plotFrontier.

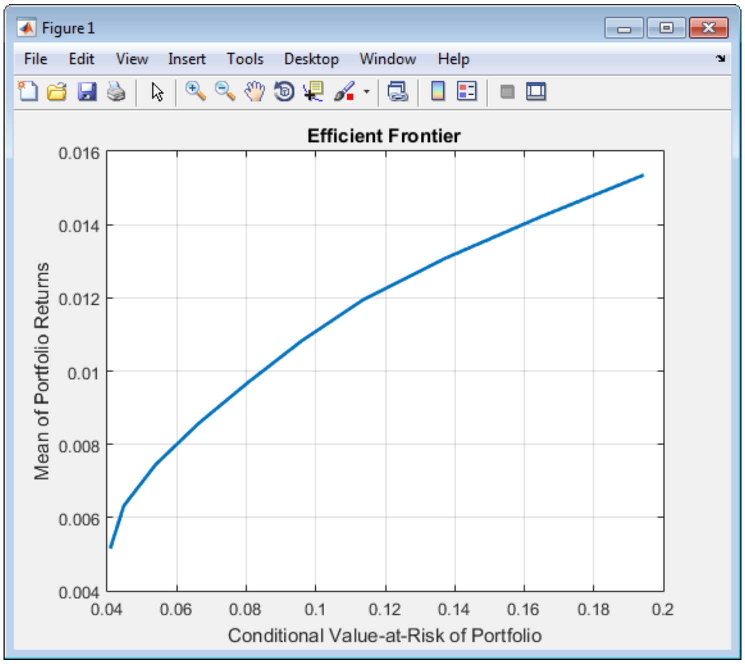
```
p.plotFrontier
```



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` object 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` object 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` object 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 object. 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 object 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 using dot notation to a `PortfolioCVaR` object. Scenarios must always be set through either the `PortfolioCVaR` object, the `setScenarios` function, or any of the scenario simulation functions.

See Also

`PortfolioCVaR` | `estimateBounds`

Related Examples

- “Common Operations on the PortfolioCVaR Object” on page 5-29
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-50
- “Hedging Using CVaR Portfolio Optimization” on page 5-118
- “Compute Maximum Reward-to-Risk Ratio for CVaR Portfolio” on page 5-130

More About

- “PortfolioCVaR Object” on page 5-17
- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-16

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- CVaR Portfolio Optimization (4 min 56 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` object:

```
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` object 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` object, 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` object:

```
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` | `setAssetList` | `setInitPort` | `estimateBounds` | `checkFeasibility`

Related Examples

- “Setting Up an Initial or Current Portfolio” on page 5-33
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-50
- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-36
- “Hedging Using CVaR Portfolio Optimization” on page 5-118
- “Compute Maximum Reward-to-Risk Ratio for CVaR Portfolio” on page 5-130

More About

- “PortfolioCVaR Object” on page 5-17

- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-16

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- CVaR Portfolio Optimization (4 min 56 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 5-79. Suppose that you have an initial portfolio in x0, then use the PortfolioCVaR object 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 object 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 5-72, “Working with One-Way Turnover Constraints Using PortfolioCVaR Object” on page 5-75, and “Working with Transaction Costs” on page 5-46. 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 object 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 | setAssetList | setInitPort | estimateBounds | checkFeasibility

Related Examples

- “Creating the PortfolioCVaR Object” on page 5-22
- “Common Operations on the PortfolioCVaR Object” on page 5-29
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-50
- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-36
- “Hedging Using CVaR Portfolio Optimization” on page 5-118
- “Compute Maximum Reward-to-Risk Ratio for CVaR Portfolio” on page 5-130

More About

- “PortfolioCVaR Object” on page 5-17
- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-16

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- CVaR Portfolio Optimization (4 min 56 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 5-5.) Therefore, the CVaR portfolio optimization problem is a stochastic optimization problem. Given a sample of scenarios, the conditional expectation that defines the sample CVaR of the portfolio can be expressed as a finite sum, a weighted average of losses. The weights of the losses depend on their relative magnitude; for a confidence level α, only the worst (1 − α) x 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-5). 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-5). For more information, see Algorithms section of `setSolver`. To learn more about the workflow when using `PortfolioCVaR` objects, see “PortfolioCVaR Object Workflow” on page 5-16.

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 object 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 object 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 ≤ `startrow` ≤ `endrow` ≤ `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 `table` or `timetable` (see “Simulating Normal Scenarios from Time Series Data” on page 5-41). 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 implicitly works with matrices of data or data in a table or timetable object using the same rules for whether the data are returns or prices. To illustrate, use array2timetable to create a timetable for 14 assets from CAPMuniverse and the use the timetable to simulate scenarios for PortfolioCVaR.

```
load CAPMuniverse
time = datetime(Dates,'ConvertFrom','datenum');
stockTT = array2timetable(Data,'RowTimes',time, 'VariableNames', Assets);
stockTT.Properties
% Notice that GOOG has missing data, because it was not listed before Aug 2004
head(stockTT, 5)
```

```
ans =
  TimetableProperties with properties:

             Description: ''
                UserData: []
          DimensionNames: {'Time'  'Variables'}
           VariableNames: {'AAPL'  'AMZN'  'CSCO'  'DELL'  'EBAY'  'GOOG'  'HPQ'  'IBM'  'INTC'
    VariableDescriptions: {}
           VariableUnits: {}
      VariableContinuity: []
                RowTimes: [1471×1 datetime]
               StartTime: 03-Jan-2000
              SampleRate: NaN
                TimeStep: NaN
        CustomProperties: No custom properties are set.
      Use addprop and rmprop to modify CustomProperties.

ans =

  5×14 timetable

       Time           AAPL         AMZN         CSCO         DELL         EBAY        GOOG       H
    ___________    _________    _________    _________    _________    _________    ____    ____

    03-Jan-2000     0.088805       0.1742     0.008775    -0.002353      0.12829    NaN       0.0
    04-Jan-2000    -0.084331     -0.08324     -0.05608     -0.08353    -0.093805    NaN      -0.07
    05-Jan-2000     0.014634     -0.14877    -0.003039     0.070984     0.066875    NaN      -0.00
    06-Jan-2000    -0.086538    -0.060072    -0.016619    -0.038847    -0.012302    NaN      -0.06
    07-Jan-2000     0.047368     0.061013       0.0587    -0.037708    -0.000964    NaN       0.02
```

Use the `'MissingData'` option offered by PortfolioCVaR to account for the missing data.

```
p = PortfolioCVaR;
p = simulateNormalScenariosByData(p, stockTT, 20000 ,'missingdata',true);
[passetmean, passetcovar] = estimateScenarioMoments(p)
```

```
passetmean =

    0.0012
    0.0007
   -0.0005
   -0.0000
    0.0016
    0.0043
   -0.0001
    0.0000
    0.0001
   -0.0002
    0.0000
    0.0004
    0.0001
    0.0001


passetcovar =

    0.0013    0.0005    0.0006    0.0005    0.0006    0.0003    0.0005    0.0003    0.0006    0.0
    0.0005    0.0024    0.0007    0.0005    0.0010    0.0005    0.0005    0.0003    0.0006    0.0
    0.0006    0.0007    0.0013    0.0006    0.0007    0.0004    0.0006    0.0004    0.0008    0.0
    0.0005    0.0005    0.0006    0.0009    0.0006    0.0002    0.0005    0.0003    0.0006    0.0
    0.0006    0.0010    0.0007    0.0006    0.0018    0.0007    0.0005    0.0003    0.0006    0.0
    0.0003    0.0005    0.0004    0.0002    0.0007    0.0013    0.0002    0.0002    0.0002    0.0
    0.0005    0.0005    0.0006    0.0005    0.0005    0.0002    0.0010    0.0003    0.0005    0.0
    0.0003    0.0003    0.0004    0.0003    0.0003    0.0002    0.0003    0.0005    0.0004    0.0
    0.0006    0.0006    0.0008    0.0006    0.0006    0.0002    0.0005    0.0004    0.0011    0.0
    0.0004    0.0004    0.0005    0.0004    0.0005    0.0002    0.0003    0.0002    0.0005    0.0
    0.0005    0.0006    0.0008    0.0005    0.0007    0.0003    0.0005    0.0004    0.0007    0.0
    0.0006    0.0011    0.0008    0.0006    0.0011    0.0011    0.0006    0.0004    0.0007    0.0
    0.0002    0.0002    0.0002    0.0002    0.0002    0.0001    0.0002    0.0002    0.0002    0.0
   -0.0000   -0.0000   -0.0000   -0.0000   -0.0000   -0.0000   -0.0000    0.0000   -0.0000   -0.0
```

Use the name-value input `'DataFormat'` to handle return or price data and `'MissingData'` to ignore or use samples with missing values. In addition, `simulateNormalScenariosByData` extracts asset names or identifiers from a `table` or `timetable` if the argument `'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 `p` from the previous example with the `'GetAssetList'` flag set to `true` extracts the column names from the timetable object:

```
p = simulateNormalScenariosByData(p, stockTT, 20000 ,'missingdata',true, 'GetAssetList', true);
disp(p.AssetList)
```

```
 'AAPL'    'AMZN'    'CSCO'    'DELL'    'EBAY'    'GOOG'    'HPQ'    'IBM'    'INTC'    'MSFT'
```

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 5-29.

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 | setCosts | setProbabilityLevel | setScenarios | estimatePortVaR | simulateNormalScenariosByMoments | simulateNormalScenariosByData

Related Examples

- “Working with a Riskless Asset” on page 5-45
- “Working with Transaction Costs” on page 5-46
- “Creating the PortfolioCVaR Object” on page 5-22
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-50
- “Validate the CVaR Portfolio Problem” on page 5-78
- “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-82
- “Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-102
- “Hedging Using CVaR Portfolio Optimization” on page 5-118
- “Compute Maximum Reward-to-Risk Ratio for CVaR Portfolio” on page 5-130

More About

- “PortfolioCVaR Object” on page 5-17
- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-16

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- CVaR Portfolio Optimization (4 min 56 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 object:

```
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 | setCosts | setProbabilityLevel | setScenarios | estimatePortVaR | simulateNormalScenariosByMoments | simulateNormalScenariosByData

Related Examples

- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-36
- “Working with Transaction Costs” on page 5-46
- “Creating the PortfolioCVaR Object” on page 5-22
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-50
- “Validate the CVaR Portfolio Problem” on page 5-78
- “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-82
- “Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-102
- “Hedging Using CVaR Portfolio Optimization” on page 5-118
- “Compute Maximum Reward-to-Risk Ratio for CVaR Portfolio” on page 5-130

More About

- “PortfolioCVaR Object” on page 5-17
- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-16

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- CVaR Portfolio Optimization (4 min 56 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 5-4). 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 object. 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 object 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 object 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 | setCosts | setProbabilityLevel | setScenarios | estimatePortVaR | simulateNormalScenariosByMoments | simulateNormalScenariosByData

Related Examples

- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-36
- “Working with a Riskless Asset” on page 5-45
- “Creating the PortfolioCVaR Object” on page 5-22
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-50
- “Validate the CVaR Portfolio Problem” on page 5-78
- “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-82

- “Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-102
- “Hedging Using CVaR Portfolio Optimization” on page 5-118
- “Compute Maximum Reward-to-Risk Ratio for CVaR Portfolio” on page 5-130

More About

- “PortfolioCVaR Object” on page 5-17
- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-16

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- CVaR Portfolio Optimization (4 min 56 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 5-16.

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 object 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: [20×1 double]
          UpperBound: []
         LowerBudget: 1
         UpperBudget: 1
         GroupMatrix: []
          LowerGroup: []
          UpperGroup: []
              GroupA: []
              GroupB: []
          LowerRatio: []
          UpperRatio: []
        MinNumAssets: []
        MaxNumAssets: []
           BoundType: [20×1 categorical]
```

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: [20×1 double]
         UpperBound: []
        LowerBudget: 1
        UpperBudget: 1
        GroupMatrix: []
         LowerGroup: []
         UpperGroup: []
             GroupA: []
             GroupB: []
         LowerRatio: []
         UpperRatio: []
       MinNumAssets: []
       MaxNumAssets: []
          BoundType: [20×1 categorical]
```

See Also

PortfolioCVaR | setDefaultConstraints | setBounds | setBudget | setGroups | setGroupRatio | setEquality | setInequality | setTurnover | setOneWayTurnover

Related Examples

- “Working with 'Simple' Bound Constraints Using PortfolioCVaR Object” on page 5-54
- “Working with Budget Constraints Using PortfolioCVaR Object” on page 5-57
- “Working with Group Constraints Using PortfolioCVaR Object” on page 5-59
- “Working with Group Ratio Constraints Using PortfolioCVaR Object” on page 5-62
- “Working with Linear Equality Constraints Using PortfolioCVaR Object” on page 5-65
- “Working with Linear Inequality Constraints Using PortfolioCVaR Object” on page 5-67
- “Working with Average Turnover Constraints Using PortfolioCVaR Object” on page 5-72
- “Working with One-Way Turnover Constraints Using PortfolioCVaR Object” on page 5-75
- “Creating the PortfolioCVaR Object” on page 5-22
- “Validate the CVaR Portfolio Problem” on page 5-78
- “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-82
- “Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-102

- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-36
- “Hedging Using CVaR Portfolio Optimization” on page 5-118
- “Compute Maximum Reward-to-Risk Ratio for CVaR Portfolio” on page 5-130

More About

- “PortfolioCVaR Object” on page 5-17
- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-16

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- CVaR Portfolio Optimization (4 min 56 sec)

Working with 'Simple' Bound Constraints Using PortfolioCVaR Object

'Simple' bound constraints are optional linear constraints that maintain upper and lower bounds on portfolio weights (see “'Simple' Bound Constraints” on page 5-9). Although every portfolio set must be bounded, it is not necessary to specify a portfolio set with explicit bound constraints. For example, you can create a portfolio set with an implicit upper bound constraint or a portfolio set with 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 4-57).

Setting 'Simple' Bounds Using the PortfolioCVaR Function

The properties for bound constraints are set through the PortfolioCVaR object. 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, 'BoundType', 'Simple');
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 4-64.

Setting 'Simple' 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,'BoundType', 'Simple');
disp(p.NumAssets)
disp(p.LowerBound)
disp(p.UpperBound)
```

```
  2

  0.5000
  0.2500

  0.7500
  0.5000
```

Setting 'Simple' Bounds Using the PortfolioCVaR Function or setBounds Function

Both the `PortfolioCVaR` object 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,'BoundType', 'Simple');
```

or

```
p = PortfolioCVaR('NumAssets', 500);
p = setBounds(p, 0, 0.05,'BoundType','Simple');
```

or

```
p = PortfolioCVaR;
p = setBounds(p, 0, 0.05,'NumAssets', 500,'BoundType','Simple');
```

To clear bound constraints from your `PortfolioCVaR` object, use either the `PortfolioCVaR` object 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` | `setDefaultConstraints` | `setBounds` | `setBudget` | `setGroups` | `setGroupRatio` | `setEquality` | `setInequality` | `setTurnover` | `setOneWayTurnover`

Related Examples

- “Creating the PortfolioCVaR Object” on page 5-22
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-50
- “Troubleshooting for Setting 'Conditional' BoundType, MinNumAssets, and MaxNumAssets Constraints” on page 4-139
- “Validate the CVaR Portfolio Problem” on page 5-78
- “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-82
- “Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-102
- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-36
- “Working with 'Conditional' BoundType, MinNumAssets, and MaxNumAssets Constraints Using PortfolioCVaR Objects” on page 5-69

- “Hedging Using CVaR Portfolio Optimization” on page 5-118
- “Compute Maximum Reward-to-Risk Ratio for CVaR Portfolio” on page 5-130

More About

- “PortfolioCVaR Object” on page 5-17
- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-16

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- CVaR Portfolio Optimization (4 min 56 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 5-10). Budget constraints have properties `LowerBudget` for the lower budget constraint and `UpperBudget` for the upper budget constraint. If you set up a CVaR portfolio optimization problem that requires portfolios to be fully invested in your universe of assets, you can set `LowerBudget` to be equal to `UpperBudget`. These budget constraints can be set with default values equal to `1` using `setDefaultConstraints` (see "Setting Default Constraints Using the PortfolioCVaR Function" on page 5-50).

Setting Budget Constraints Using the PortfolioCVaR Function

The properties for the budget constraint can also be set using the `PortfolioCVaR` object. 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 5-45. To clear either bound for the budget constraint from your `PortfolioCVaR` object, use either the `PortfolioCVaR` object 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` | `setDefaultConstraints` | `setBounds` | `setBudget` | `setGroups` | `setGroupRatio` | `setEquality` | `setInequality` | `setTurnover` | `setOneWayTurnover`

Related Examples

- “Creating the PortfolioCVaR Object” on page 5-22
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-50
- “Validate the CVaR Portfolio Problem” on page 5-78
- “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-82
- “Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-102
- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-36
- “Hedging Using CVaR Portfolio Optimization” on page 5-118
- “Compute Maximum Reward-to-Risk Ratio for CVaR Portfolio” on page 5-130

More About

- “PortfolioCVaR Object” on page 5-17
- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-16

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- CVaR Portfolio Optimization (4 min 56 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 5-11). Although the constraints are implemented as general constraints, the usual convention is to form a group matrix that identifies membership of each asset within a specific group with Boolean indicators (either `true` or `false` or with `1` or `0`) for each element in the group matrix. Group constraints have properties `GroupMatrix` for the group membership matrix, `LowerGroup` for the lower-bound constraint on groups, and `UpperGroup` for the upper-bound constraint on groups.

Setting Group Constraints Using the PortfolioCVaR Function

The properties for group constraints are set through the `PortfolioCVaR` object. 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
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` object, `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` | `setDefaultConstraints` | `setBounds` | `setBudget` | `setGroups` | `setGroupRatio` | `setEquality` | `setInequality` | `setTurnover` | `setOneWayTurnover`

Related Examples

- “Creating the PortfolioCVaR Object” on page 5-22
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-50
- “Validate the CVaR Portfolio Problem” on page 5-78
- “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-82
- “Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-102
- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-36
- “Hedging Using CVaR Portfolio Optimization” on page 5-118
- “Compute Maximum Reward-to-Risk Ratio for CVaR Portfolio” on page 5-130

More About

- “PortfolioCVaR Object” on page 5-17
- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-16

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- CVaR Portfolio Optimization (4 min 56 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 5-11). 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` object. 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 nonfinancial companies (assets 4–6). To set group ratio constraints:

```
GA = [ 1 1 1 0 0 0 ];    % financial companies
GB = [ 0 0 0 1 1 1 ];    % nonfinancial companies
p = PortfolioCVaR('GroupA', GA, 'GroupB', GB, 'UpperRatio', 0.5);
disp(p.NumAssets)
disp(p.GroupA)
disp(p.GroupB)
disp(p.UpperRatio)
```

```
6

1     1     1     0     0     0

0     0     0     1     1     1

0.5000
```

Group matrices `GA` and `GB` in this example can be logical matrices with `true` and `false` elements that yield the same result:

```
GA = [ true true true false false false ];    % financial companies
GB = [ false false false true true true ];    % nonfinancial companies
p = PortfolioCVaR('GroupA', GA, 'GroupB', GB, 'UpperRatio', 0.5);
disp(p.NumAssets)
disp(p.GroupA)
disp(p.GroupB)
disp(p.UpperRatio)
```

```
6

1     1     1     0     0     0

0     0     0     1     1     1

0.5000
```

Setting Group Ratio Constraints Using the setGroupRatio and addGroupRatio 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 nonfinancial companies (assets 4–6). Given a `PortfolioCVaR` object `p`, use `setGroupRatio` to set the group constraints:

```
GA = [ true true true false false false ];   % financial companies
GB = [ false false false true true true ];   % nonfinancial companies
p = PortfolioCVaR;
p = setGroupRatio(p, GA, GB, [], 0.5);
disp(p.NumAssets)
disp(p.GroupA)
disp(p.GroupB)
disp(p.UpperRatio)
```

```
6

1     1     1     0     0     0

0     0     0     1     1     1

0.5000
```

In this example, you would set the `LowerRatio` property to be empty (`[]`).

Suppose 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` object, `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` | `setDefaultConstraints` | `setBounds` | `setBudget` | `setGroups` | `setGroupRatio` | `setEquality` | `setInequality` | `setTurnover` | `setOneWayTurnover`

Related Examples

- “Creating the PortfolioCVaR Object” on page 5-22
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-50
- “Validate the CVaR Portfolio Problem” on page 5-78
- “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-82
- “Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-102
- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-36
- “Hedging Using CVaR Portfolio Optimization” on page 5-118
- “Compute Maximum Reward-to-Risk Ratio for CVaR Portfolio” on page 5-130

More About

- “PortfolioCVaR Object” on page 5-17
- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-16

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- CVaR Portfolio Optimization (4 min 56 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 5-9). Linear equality constraints have properties `AEquality`, for the equality constraint matrix, and `bEquality`, for the equality constraint vector.

Setting Linear Equality Constraints Using the PortfolioCVaR Function

The properties for linear equality constraints are set using the `PortfolioCVaR` object. 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` object, `setEquality`, and `addEquality` implement scalar expansion on the `bEquality` property based on the dimension of the matrix in the `AEquality` property.

See Also

`PortfolioCVaR` | `setDefaultConstraints` | `setBounds` | `setBudget` | `setGroups` | `setGroupRatio` | `setEquality` | `setInequality` | `setTurnover` | `setOneWayTurnover`

Related Examples

- “Creating the PortfolioCVaR Object” on page 5-22
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-50
- “Validate the CVaR Portfolio Problem” on page 5-78
- “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-82
- “Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-102
- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-36
- “Hedging Using CVaR Portfolio Optimization” on page 5-118
- “Compute Maximum Reward-to-Risk Ratio for CVaR Portfolio” on page 5-130

More About

- “PortfolioCVaR Object” on page 5-17
- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-16

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- CVaR Portfolio Optimization (4 min 56 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 5-8). Linear inequality constraints have properties `AInequality` for the inequality constraint matrix, and `bInequality` for the inequality constraint vector.

Setting Linear Inequality Constraints Using the PortfolioCVaR Function

The properties for linear inequality constraints are set using the `PortfolioCVaR` object. 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` object, `setInequality`, and `addInequality` implement scalar expansion on the `bInequality` property based on the dimension of the matrix in the `AInequality` property.

See Also

`PortfolioCVaR` | `setDefaultConstraints` | `setBounds` | `setBudget` | `setGroups` | `setGroupRatio` | `setEquality` | `setInequality` | `setTurnover` | `setOneWayTurnover`

Related Examples

- “Creating the PortfolioCVaR Object” on page 5-22
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-50
- “Validate the CVaR Portfolio Problem” on page 5-78
- “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-82
- “Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-102
- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-36
- “Hedging Using CVaR Portfolio Optimization” on page 5-118
- “Compute Maximum Reward-to-Risk Ratio for CVaR Portfolio” on page 5-130

More About

- “PortfolioCVaR Object” on page 5-17
- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-16

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- CVaR Portfolio Optimization (4 min 56 sec)

Working with 'Conditional' BoundType, MinNumAssets, and MaxNumAssets Constraints Using PortfolioCVaR Objects

When any one, or any combination of `'Conditional'` `BoundType`, `MinNumAssets`, or `MaxNumAssets` constraints are active, the portfolio problem is formulated by adding `NumAssets` binary variables, where `0` indicates not invested, and `1` is invested. For example, to explain the `'Conditional'` `BoundType` and `MinNumAssets` and `MaxNumAssets` constraints, assume that your portfolio has a universe of 100 assets that you want to invest:

- `'Conditional'` `BoundType` (also known as semicontinuous constraints), set by `setBounds`, is often used in situations where you do not want to invest small values. A standard example is a portfolio optimization problem where many small allocations are not attractive because of transaction costs. Instead, you prefer fewer instruments in the portfolio with larger allocations. This situation can be handled using`'Conditional'` `BoundType` constraints for a `PortfolioCVaR` object.

  For example, the weight you invest in each asset is either `0` or between `[0.01, 0.5]`. Generally, a semicontinuous variable *x* is a continuous variable between bounds [`lb`, `ub`] that also can assume the value `0`, where `lb` > `0`, `lb` ≤ `ub`. Applying this to portfolio optimization requires that very small or large positions should be avoided, that is values that fall in (`0`, `lb`) or are more than `ub`.
- `MinNumAssets` and `MaxNumAssets` (also known as cardinality constraints), set by `setMinMaxNumAssets`, limit the number of assets in a `PortfolioCVaR` object. For example, if you have 100 assets in your portfolio and you want the number of assets allocated in the portfolio to be from 40 through 60. Using `MinNumAssets` and `MaxNumAssets` you can limit the number of assets in the optimized portfolio, which allows you to limit transaction and operational costs or to create an index tracking portfolio.

Setting 'Conditional' BoundType Constraints Using the setBounds Function

Use `setBounds` with a `'conditional'` `BoundType` to set *xi* = `0` or `0.02` <= *xi* <= `0.5` for all *i*=1,...`NumAssets`:

```
p = PortfolioCVaR;
p = setBounds(p, 0.02, 0.5,'BoundType', 'Conditional', 'NumAssets', 3)
```

```
p = 

  PortfolioCVaR with properties:

             BuyCost: []
            SellCost: []
        RiskFreeRate: []
    ProbabilityLevel: []
            Turnover: []
         BuyTurnover: []
        SellTurnover: []
        NumScenarios: []
                Name: []
           NumAssets: 3
           AssetList: []
            InitPort: []
         AInequality: []
         bInequality: []
           AEquality: []
           bEquality: []
          LowerBound: [3×1 double]
          UpperBound: [3×1 double]
         LowerBudget: []
         UpperBudget: []
         GroupMatrix: []
          LowerGroup: []
          UpperGroup: []
              GroupA: []
              GroupB: []
          LowerRatio: []
          UpperRatio: []
        MinNumAssets: []
        MaxNumAssets: []
           BoundType: [3×1 categorical]
```

Setting the Limits on the Number of Assets Invested Using the setMinMaxNumAssets Function

You can also set the MinNumAssets and MaxNumAssets properties to define a limit on the number of assets invested using setMinMaxNumAssets. For example, by setting MinNumAssets=MaxNumAssets=2, only two of the three assets are invested in the portfolio.

```
p = PortfolioCVaR;
p = setBounds(p, 0.02, 0.5,'BoundType', 'Conditional', 'NumAssets', 3)
p = setMinMaxNumAssets(p, 2, 2)
```

```
 p = 

  PortfolioCVaR with properties:

             BuyCost: []
            SellCost: []
        RiskFreeRate: []
    ProbabilityLevel: []
            Turnover: []
         BuyTurnover: []
        SellTurnover: []
        NumScenarios: []
                Name: []
           NumAssets: 3
           AssetList: []
            InitPort: []
         AInequality: []
         bInequality: []
           AEquality: []
           bEquality: []
          LowerBound: [3×1 double]
          UpperBound: [3×1 double]
         LowerBudget: []
         UpperBudget: []
         GroupMatrix: []
          LowerGroup: []
          UpperGroup: []
```

```
             GroupA: []
             GroupB: []
         LowerRatio: []
         UpperRatio: []
       MinNumAssets: 2
       MaxNumAssets: 2
          BoundType: [3×1 categorical]
```

See Also

`PortfolioCVaR` | `setBounds` | `setMinMaxNumAssets` | `setDefaultConstraints` | `setBounds` | `setBudget` | `setGroups` | `setGroupRatio` | `setEquality` | `setInequality` | `setTurnover` | `setOneWayTurnover`

Related Examples

- "Creating the PortfolioCVaR Object" on page 5-22
- "Working with CVaR Portfolio Constraints Using Defaults" on page 5-50
- "Working with 'Simple' Bound Constraints Using PortfolioCVaR Object" on page 5-54
- "Troubleshooting for Setting 'Conditional' BoundType, MinNumAssets, and MaxNumAssets Constraints" on page 4-139
- "Validate the CVaR Portfolio Problem" on page 5-78
- "Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object" on page 5-82
- "Estimate Efficient Frontiers for PortfolioCVaR Object" on page 5-102
- "Asset Returns and Scenarios Using PortfolioCVaR Object" on page 5-36
- "Hedging Using CVaR Portfolio Optimization" on page 5-118
- "Compute Maximum Reward-to-Risk Ratio for CVaR Portfolio" on page 5-130

More About

- "PortfolioCVaR Object" on page 5-17
- "Portfolio Optimization Theory" on page 5-3
- "PortfolioCVaR Object Workflow" on page 5-16

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- CVaR Portfolio Optimization (4 min 56 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 5-12) that enforces an upper bound on the average of purchases and sales. The turnover constraint can be set using the `PortfolioCVaR` object 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` object. 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 5-33).

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` object or `setTurnover` with empty inputs for the properties to be cleared.

See Also

`PortfolioCVaR` | `setDefaultConstraints` | `setBounds` | `setBudget` | `setGroups` | `setGroupRatio` | `setEquality` | `setInequality` | `setTurnover` | `setOneWayTurnover`

Related Examples

- “Creating the PortfolioCVaR Object” on page 5-22
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-50
- “Validate the CVaR Portfolio Problem” on page 5-78

- “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-82
- “Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-102
- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-36
- “Hedging Using CVaR Portfolio Optimization” on page 5-118
- “Compute Maximum Reward-to-Risk Ratio for CVaR Portfolio” on page 5-130

More About

- “PortfolioCVaR Object” on page 5-17
- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-16

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- CVaR Portfolio Optimization (4 min 56 sec)

Working with One-Way Turnover Constraints Using PortfolioCVaR Object

One-way turnover constraints are optional constraints (see “One-way Turnover Constraints” on page 5-13) that enforce upper bounds on net purchases or net sales. One-way turnover constraints can be set using the PortfolioCVaR object 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 object. 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 5-33).

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 thePortfolioCVaR object or setOneWayTurnover with empty inputs for the properties to be cleared.

See Also

`PortfolioCVaR` | `setDefaultConstraints` | `setBounds` | `setBudget` | `setGroups` | `setGroupRatio` | `setEquality` | `setInequality` | `setTurnover` | `setOneWayTurnover`

Related Examples

- “Creating the PortfolioCVaR Object” on page 5-22
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-50
- “Validate the CVaR Portfolio Problem” on page 5-78
- “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-82
- “Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-102
- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-36
- “Hedging Using CVaR Portfolio Optimization” on page 5-118
- “Compute Maximum Reward-to-Risk Ratio for CVaR Portfolio” on page 5-130

More About

- “PortfolioCVaR Object” on page 5-17
- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-16

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- CVaR Portfolio Optimization (4 min 56 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 5-16.

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` | `estimateBounds` | `checkFeasibility`

Related Examples

- “Creating the PortfolioCVaR Object” on page 5-22
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-50
- “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-82
- “Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-102
- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-36

- “Hedging Using CVaR Portfolio Optimization” on page 5-118
- “Compute Maximum Reward-to-Risk Ratio for CVaR Portfolio” on page 5-130

More About

- “PortfolioCVaR Object” on page 5-17
- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-16

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- CVaR Portfolio Optimization (4 min 56 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. The “Obtaining Portfolios Along the Entire Efficient Frontier” on page 5-83 example focuses on the former goal and “Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-102 focuses on the latter goal. For information on the workflow when using PortfolioCVaR objects, see “PortfolioCVaR Object Workflow” on page 5-16.

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 an input argument 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)
```

```
    0.8645    0.6960    0.5319    0.3649    0.1948    0.0215         0         0         0
    0.0602    0.1582    0.2551    0.3559    0.4660    0.5740    0.4573    0.3115    0.1599
    0.0326    0.0451    0.0503    0.0567    0.0577    0.0677    0.0501    0.0253    0.0077
    0.0428    0.1007    0.1627    0.2226    0.2815    0.3368    0.4926    0.6632    0.8325    1.0
```

If you want only four portfolios, you can use `estimateFrontier` with `NumPorts` specified as `4`.

```
pwgt = estimateFrontier(p, 4);
disp(pwgt)
```

```
    0.8645    0.3649         0         0
    0.0602    0.3559    0.4573         0
    0.0326    0.0567    0.0501         0
    0.0428    0.2226    0.4926    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)
```

```
pwgt = 4×10
```

```
    0.8645    0.6960    0.5319    0.3649    0.1948    0.0215         0         0         0
    0.0602    0.1582    0.2551    0.3559    0.4660    0.5740    0.4573    0.3115    0.1599
    0.0326    0.0451    0.0503    0.0567    0.0577    0.0677    0.0501    0.0253    0.0077
    0.0428    0.1007    0.1627    0.2226    0.2815    0.3368    0.4926    0.6632    0.8325    1.0
```

```
display(pbuy)
```

```
pbuy = 4×10

    0.5645    0.3960    0.2319    0.0649         0         0         0         0         0
         0         0         0    0.0559    0.1660    0.2740    0.1573    0.0115         0
         0         0         0         0         0         0         0         0         0
         0    0.0007    0.0627    0.1226    0.1815    0.2368    0.3926    0.5632    0.7325    0.9
```

```
display(psell)
```

```
psell = 4×10

         0         0         0         0    0.1052    0.2785    0.3000    0.3000    0.3000    0.3
    0.2398    0.1418    0.0449         0         0         0         0         0    0.1401    0.3
    0.1674    0.1549    0.1497    0.1433    0.1423    0.1323    0.1499    0.1747    0.1923    0.2
    0.0572         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

`PortfolioCVaR` | `estimateFrontier` | `estimateFrontierLimits` | `estimateFrontierByReturn` | `estimatePortReturn` | `estimateFrontierByRisk` | `estimatePortRisk` | `estimateFrontierByRisk` | `setSolver`

Related Examples

- “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-82
- “Creating the PortfolioCVaR Object” on page 5-22
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-50
- “Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-102
- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-36
- “Troubleshooting CVaR Portfolio Optimization Results” on page 5-115
- “Hedging Using CVaR Portfolio Optimization” on page 5-118
- “Compute Maximum Reward-to-Risk Ratio for CVaR Portfolio” on page 5-130

More About

- “PortfolioCVaR Object” on page 5-17
- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-16

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- CVaR Portfolio Optimization (4 min 56 sec)

Obtaining Endpoints of the Efficient Frontier

Often when using a PortfolioCVaR object, 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)
```

```
    0.8645         0
    0.0602         0
    0.0326         0
    0.0428    1.0000
```

The estimatePortMoments function shows the range of risks and returns for efficient portfolios: Note that 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)

pwgt = 4×2

    0.8671         0
    0.0521         0
    0.0419         0
    0.0388    1.0000
```

```
display(pbuy)

pbuy = 4×2

    0.5671         0
         0         0
         0         0
         0    0.9000
```

```
display(psell)

psell = 4×2

         0    0.3000
    0.2479    0.3000
    0.1581    0.2000
    0.0612         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` | `estimateFrontierLimits` | `estimateFrontierByReturn` | `estimatePortReturn` | `estimateFrontierByRisk` | `estimatePortRisk` | `estimateFrontierByRisk` | `setSolver`

Related Examples

- “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-82
- “Creating the PortfolioCVaR Object” on page 5-22
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-50
- “Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-102
- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-36
- “Troubleshooting CVaR Portfolio Optimization Results” on page 5-115
- “Hedging Using CVaR Portfolio Optimization” on page 5-118
- “Compute Maximum Reward-to-Risk Ratio for CVaR Portfolio” on page 5-130

More About

- “PortfolioCVaR Object” on page 5-17
- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-16

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- CVaR Portfolio Optimization (4 min 56 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 5-86 and “Obtaining CVaR Portfolio Risks and Returns” on page 5-102).

```
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` | `estimateFrontierLimits` | `estimateFrontierByReturn` | `estimatePortReturn` | `estimateFrontierByRisk` | `estimatePortRisk` | `estimateFrontierByRisk` | `setSolver`

Related Examples

- “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-82
- “Creating the PortfolioCVaR Object” on page 5-22
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-50
- “Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-102
- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-36
- “Troubleshooting CVaR Portfolio Optimization Results” on page 5-115
- “Hedging Using CVaR Portfolio Optimization” on page 5-118
- “Compute Maximum Reward-to-Risk Ratio for CVaR Portfolio” on page 5-130

More About

- “PortfolioCVaR Object” on page 5-17
- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-16

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- CVaR Portfolio Optimization (4 min 56 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 5-86 and “Obtaining CVaR Portfolio Risks and Returns” on page 5-102).

```
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` | `estimateFrontierLimits` | `estimateFrontierByReturn` | `estimatePortReturn` | `estimateFrontierByRisk` | `estimatePortRisk` | `estimateFrontierByRisk` | `setSolver`

Related Examples

- “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-82
- “Creating the PortfolioCVaR Object” on page 5-22
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-50

- “Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-102
- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-36
- “Troubleshooting CVaR Portfolio Optimization Results” on page 5-115
- “Hedging Using CVaR Portfolio Optimization” on page 5-118
- “Compute Maximum Reward-to-Risk Ratio for CVaR Portfolio” on page 5-130

More About

- “PortfolioCVaR Object” on page 5-17
- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-16

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- CVaR Portfolio Optimization (4 min 56 sec)

Choosing and Controlling the Solver for PortfolioCVaR Optimizations

When solving portfolio optimizations for a `PortfolioCVaR` object, you are solving nonlinear optimization problems with either nonlinear objective or nonlinear constraints. You can use `'TrustRegionCP'` (default), `'ExtendedCP'`, or `'cuttingplane'` solvers that implement Kelley’s cutting plane method (see Kelley [45] at “Portfolio Optimization” on page A-5). Alternatively, you can use `fmincon` and all variations of `fmincon` from Optimization Toolbox are supported. When using `fmincon` as the `solverType`, `'sqp'` is the default algorithm for `fmincon`.

Using 'TrustRegionCP', 'ExtendedCP', and 'cuttingplane' SolverTypes

The `'TrustRegionCP'`, `'ExtendedCP'`, and `'cuttingplane'` solvers have options to control the number iterations and stopping tolerances. Moreover, these solvers use `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 `'TrustRegionCP'`:

```
p = PortfolioCVaR;
p = setSolver(p, 'TrustRegionCP', 'MaxIterations', 2000);
display(p.solverType)
display(p.solverOptions)
```

```
trustregioncp
                MaxIterations: 2000
         AbsoluteGapTolerance: 1.0000e-07
         RelativeGapTolerance: 1.0000e-05
       NonlinearScalingFactor: 1000
       ObjectiveScalingFactor: 1000
          MainSolverOptions: [1×1 optim.options.Linprog]
                      Display: 'off'
                CutGeneration: 'basic'
     MaxIterationsInactiveCut: 30
           ActiveCutTolerance: 1.0000e-07
                  ShrinkRatio: 0.7500
    TrustRegionStartIteration: 2
                   DeltaLimit: 1
```

To change the main solver algorithm to `'interior-point'`, with no display, use `setSolver` to modify `'MainSolverOptions'`:

```
p = PortfolioCVaR;
options = optimoptions('linprog','Algorithm','interior-point','Display','off');
p = setSolver(p,'TrustRegionCP','MainSolverOptions',options);
display(p.solverType)
display(p.solverOptions)
display(p.solverOptions.MainSolverOptions.Algorithm)
display(p.solverOptions.MainSolverOptions.Display)
```

```
trustregioncp
                MaxIterations: 1000
         AbsoluteGapTolerance: 1.0000e-07
         RelativeGapTolerance: 1.0000e-05
       NonlinearScalingFactor: 1000
       ObjectiveScalingFactor: 1000
          MaainSolverOptions: [1×1 optim.options.Linprog]
```

```
                      Display: 'off'
                CutGeneration: 'basic'
     MaxIterationsInactiveCut: 30
           ActiveCutTolerance: 1.0000e-07
                  ShrinkRatio: 0.7500
    TrustRegionStartIteration: 2
                   DeltaLimit: 1

interior-point
off
```

Using 'fmincon' SolverType

Unlike Optimization Toolbox which uses the `interior-point` algorithm as the default algorithm for `fmincon`, the portfolio optimization for a `PortfolioCVaR` object uses the `sqp` algorithm. For details about `fmincon` and constrained nonlinear optimization algorithms and options, see “Constrained Nonlinear Optimization Algorithms”.

To modify `fmincon` options for CVaR portfolio optimizations, use `setSolver` to set the hidden properties `solverType` and `solverOptions` to specify and control the solver. (Note that you can see the default options by creating a dummy `PortfolioCVaR` object, using `p = PortfolioCVaR` and then type `p.solverOptions`.) Since these solver properties are hidden, you cannot set them using the `PortfolioCVaR` object. The default for the `fmincon` solver is to use the `sqp` algorithm objective function, gradients turned on, and no displayed output, so you do not need to use `setSolver` to specify the `sqp` algorithm.

```
p = PortfolioCVaR;
p = setSolver(p, 'fmincon');
display(p.solverOptions)
```

```
fmincon options:

   Options used by current Algorithm ('sqp'):
   (Other available algorithms: 'active-set', 'interior-point', 'sqp-legacy', 'trust-region-refle

   Set properties:
                    Algorithm: 'sqp'
          ConstraintTolerance: 1.0000e-08
                      Display: 'off'
          OptimalityTolerance: 1.0000e-08
    SpecifyConstraintGradient: 1
     SpecifyObjectiveGradient: 1
                StepTolerance: 1.0000e-08

   Default properties:
               CheckGradients: 0
     FiniteDifferenceStepSize: 'sqrt(eps)'
         FiniteDifferenceType: 'forward'
       MaxFunctionEvaluations: '100*numberOfVariables'
                MaxIterations: 400
               ObjectiveLimit: -1.0000e+20
                    OutputFcn: []
                      PlotFcn: []
                 ScaleProblem: 0
                     TypicalX: 'ones(numberOfVariables,1)'
                  UseParallel: 0
```

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 `'active-set'` algorithm and with no displayed output, use `setSolver` with:

```
p = PortfolioCVaR;
p = setSolver(p, 'fmincon','Algorithm','active-set','Display','off');
display(p.solverOptions.Algorithm)
display(p.solverOptions.Display)
```

```
active-set
off
```

Alternatively, `setSolver` accepts an `optimoptions` object from Optimization Toolbox 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
```

Using the Mixed Integer Nonlinear Programming (MINLP) Solver

The mixed integer nonlinear programming (MINLP) solver, configured using `setSolverMINLP`, enables you to specify associated solver options for portfolio optimization for a `PortfolioCVaR` object. The MINLP solver is used when any one, or any combination of `'Conditional'` `BoundType`, `MinNumAssets`, or `MaxNumAssets` constraints are active. In this case, the portfolio problem is formulated by adding `NumAssets` binary variables, where `0` indicates not invested, and `1` is invested. For more information on using `'Conditional'` `BoundType`, see `setBounds`. For more information on specifying `MinNumAssets` and `MaxNumAssets`, see `setMinMaxNumAssets`.

When using the `estimate` functions with a `PortfolioCVaR` object where `'Conditional'` `BoundType`, `MinNumAssets`, or `MaxNumAssets` constraints are active, the mixed integer nonlinear programming (MINLP) solver is automatically used.

Solver Guidelines for PortfolioCVaR Objects

The following table provides guidelines for using `setSolver` and `setSolverMINLP`.

PortfolioCVaR Problem	PortfolioCVaR Function	Type of Optimization Problem	Main Solver	Helper Solver
PortfolioCVaR without active `'Conditional'` `BoundType`, `MinNumAssets`, and `MaxNumAssets`	`estimateFrontierByRisk`	Optimizing a portfolio for a certain risk level introduces a nonlinear constraint. Therefore, this problem has a linear objective with linear and nonlinear constraints.	`'TrustRegionCP'`, `'ExtendedCP'`, `'fmincon'`, or `'cuttingplane'` using `setSolver`	`'linprog'` using `setSolver`
PortfolioCVaR without active `'Conditional'` `BoundType`, `MinNumAssets`, and `MaxNumAssets`	`estimateFrontierByReturn`	Nonlinear objective with linear constraints	`'TrustRegionCP'`, `'ExtendedCP'`, `'fmincon'`, or `'cuttingplane'` using `setSolver`	`'linprog'` using `setSolver`
PortfolioCVaR without active `'Conditional'` `BoundType`, `MinNumAssets`, and `MaxNumAssets`	`estimateFrontierLimits`	Nonlinear or linear objective with linear constraints	For ‘`min`’: nonlinear objective, `'TrustRegionCP'`, `'ExtendedCP'`, `'fmincon'`, or `'cuttingplane'` using `setSolver` For ‘`max`’: linear objective, `'linprog'` using `setSolver`	Not applicable

PortfolioCVaR Problem	PortfolioCVaR Function	Type of Optimization Problem	Main Solver	Helper Solver
PortfolioCVaR with active 'Conditional' BoundType, MinNumAssets, and MaxNumAssets	estimateFrontierByRisk	The problem is formulated by introducing NumAssets binary variables to indicate whether the corresponding asset is invested or not. Therefore, it requires a mixed integer nonlinear programming solver. Three types of MINLP solvers are offered, see setSolverMINLP.	Mixed integer nonlinear programming solver (MINLP) using setSolverMINLP	'fmincon' is used when the estimate functions reduce the problem into NLP. This solver is configured through setSolver.
PortfolioCVaR with active 'Conditional' BoundType, MinNumAssets, and MaxNumAssets	estimateFrontierByReturn	The problem is formulated by introducing NumAssets binary variables to indicate whether the corresponding asset is invested or not. Therefore, it requires a mixed integer nonlinear programming solver. Three types of MINLP solvers are offered, see setSolverMINLP.	Mixed integer nonlinear programming solver (MINLP) using setSolverMINLP	'fmincon' is used when the estimate functions reduce the problem into NLP. This solver is configured through setSolver

PortfolioCVaR Problem	PortfolioCVaR Function	Type of Optimization Problem	Main Solver	Helper Solver
PortfolioCVaR with active `'Conditional'` `BoundType`, `MinNumAssets`, and `MaxNumAssets`	`estimateFrontierLimits`	The problem is formulated by introducing `NumAssets` binary variables to indicate whether the corresponding asset is invested or not. Therefore, it requires a mixed integer nonlinear programming solver. Three types of MINLP solvers are offered, see `setSolverMINLP`.	Mixed integer nonlinear programming solver (MINLP) using `setSolverMINLP`	`'fmincon'` is used when the `estimate` functions reduce the problem into NLP. This solver is configured through `setSolver`

See Also

`PortfolioCVaR` | `estimateFrontier` | `estimateFrontierLimits` | `estimateFrontierByReturn` | `estimatePortReturn` | `estimateFrontierByRisk` | `estimatePortRisk` | `estimateFrontierByRisk` | `setSolver` | `setSolverMINLP`

Related Examples

- “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-82
- “Creating the PortfolioCVaR Object” on page 5-22
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-50
- “Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-102
- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-36
- “Troubleshooting CVaR Portfolio Optimization Results” on page 5-115
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-183
- “Hedging Using CVaR Portfolio Optimization” on page 5-118
- “Compute Maximum Reward-to-Risk Ratio for CVaR Portfolio” on page 5-130

More About

- “PortfolioCVaR Object” on page 5-17
- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-16
- “Working with 'Conditional' BoundType, MinNumAssets, and MaxNumAssets Constraints Using Portfolio Objects” on page 4-78

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- CVaR Portfolio Optimization (4 min 56 sec)

Estimate Efficient Frontiers for PortfolioCVaR Object

Whereas "Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object" on page 5-82 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 5-16.

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 | plotFrontier | estimatePortStd | estimatePortVaR

Related Examples

- “Plotting the Efficient Frontier for a PortfolioCVaR Object” on page 5-105
- “Creating the PortfolioCVaR Object” on page 5-22
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-50
- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-36
- “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-82
- “Postprocessing Results to Set Up Tradable Portfolios” on page 5-110
- “Hedging Using CVaR Portfolio Optimization” on page 5-118
- “Compute Maximum Reward-to-Risk Ratio for CVaR Portfolio” on page 5-130

More About

- “PortfolioCVaR Object” on page 5-17
- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-16

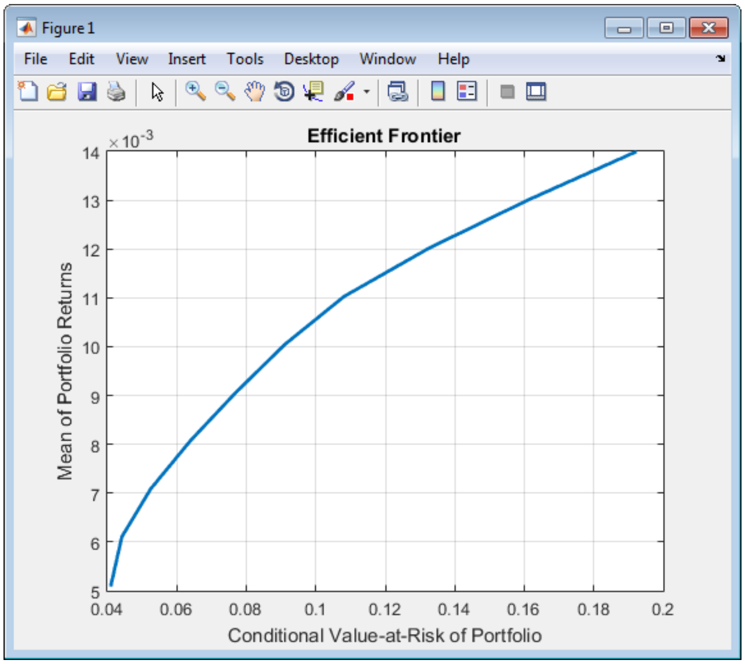
External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- CVaR Portfolio Optimization (4 min 56 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:

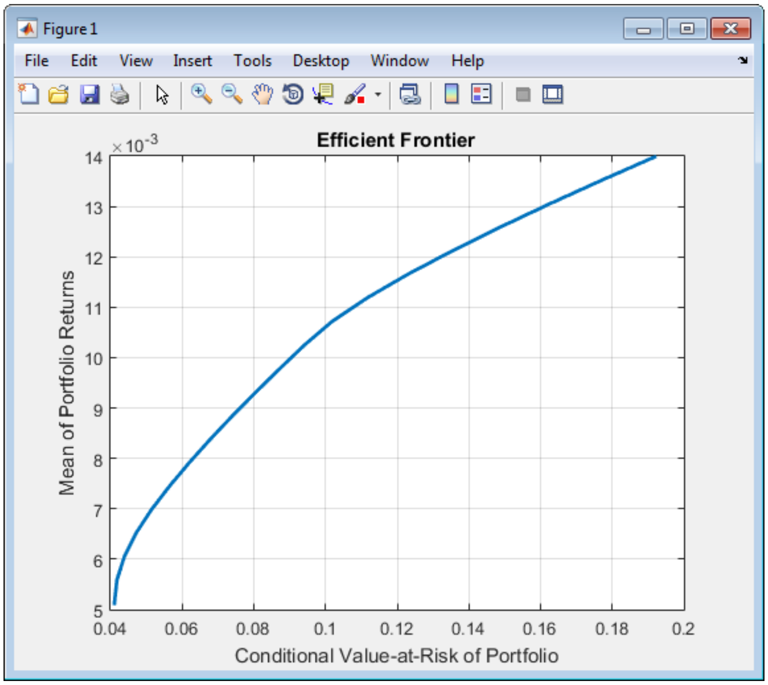
```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

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])
```

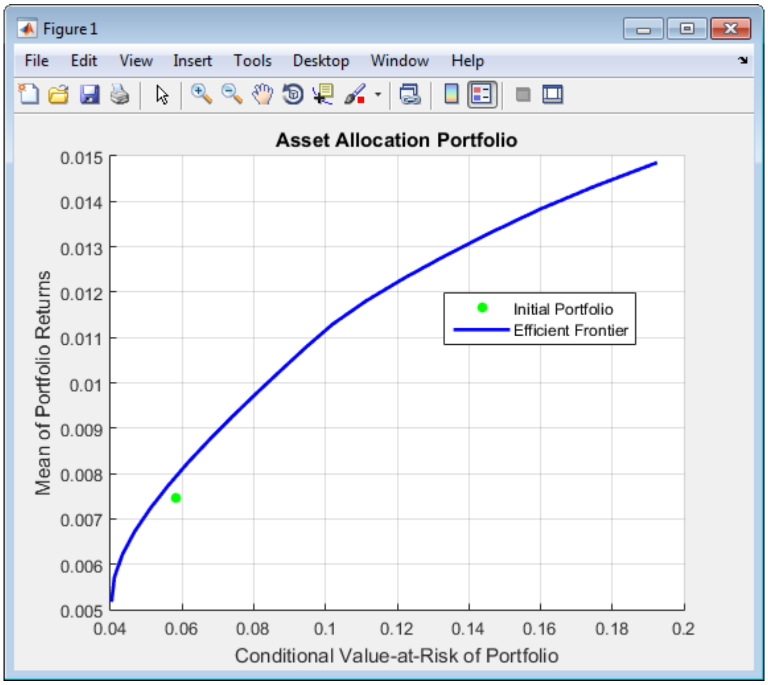
```
ans =

    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 5-82), 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)
```

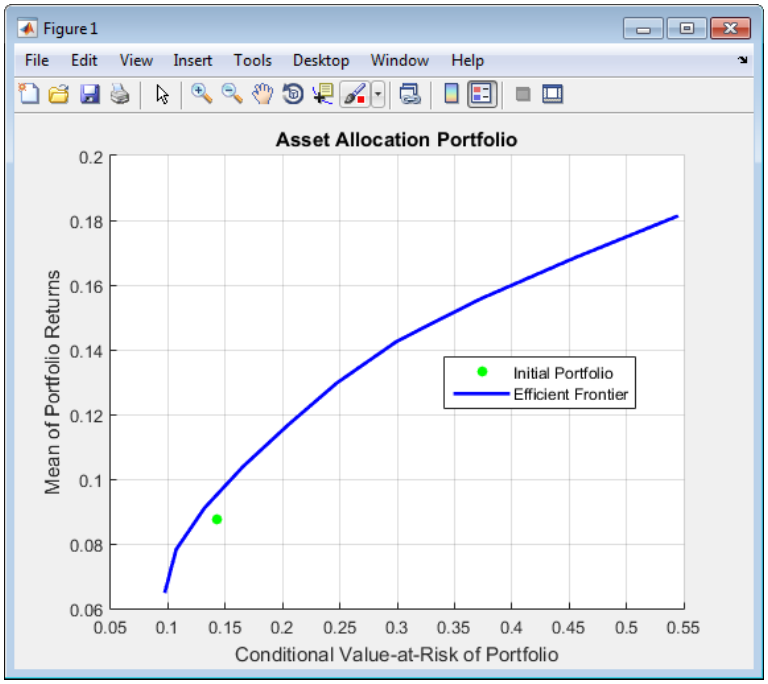


Plotting Existing Efficient Portfolio Risks and Returns

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 | plotFrontier | estimatePortStd | estimatePortVaR

Related Examples

- “Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-102
- “Creating the PortfolioCVaR Object” on page 5-22
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-50
- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-36
- “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-82
- “Postprocessing Results to Set Up Tradable Portfolios” on page 5-110
- “Hedging Using CVaR Portfolio Optimization” on page 5-118

- “Compute Maximum Reward-to-Risk Ratio for CVaR Portfolio” on page 5-130

More About

- “PortfolioCVaR Object” on page 5-17
- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-16

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- CVaR Portfolio Optimization (4 min 56 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 5-16.

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` | `estimateScenarioMoments` | `checkFeasibility`

Related Examples

- “Troubleshooting CVaR Portfolio Optimization Results” on page 5-115
- “Creating the PortfolioCVaR Object” on page 5-22
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-50
- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-36
- “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-82
- “Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-102
- “Hedging Using CVaR Portfolio Optimization” on page 5-118
- “Compute Maximum Reward-to-Risk Ratio for CVaR Portfolio” on page 5-130

More About

- “PortfolioCVaR Object” on page 5-17
- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-16

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- CVaR Portfolio Optimization (4 min 56 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

PortfolioCVaR | Portfolio

Related Examples

- "Creating the Portfolio Object" on page 4-24
- "Creating the PortfolioCVaR Object" on page 5-22
- "Working with CVaR Portfolio Constraints Using Defaults" on page 5-50
- "Asset Returns and Scenarios Using PortfolioCVaR Object" on page 5-36
- "Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object" on page 5-82
- "Estimate Efficient Frontiers for PortfolioCVaR Object" on page 5-102
- "Hedging Using CVaR Portfolio Optimization" on page 5-118
- "Compute Maximum Reward-to-Risk Ratio for CVaR Portfolio" on page 5-130

More About

- "PortfolioCVaR Object" on page 5-17
- "Portfolio Optimization Theory" on page 5-3
- "PortfolioCVaR Object Workflow" on page 5-16
- "Portfolio Object Workflow" on page 4-17

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- CVaR Portfolio Optimization (4 min 56 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` object if you want to modify it, otherwise it creates a new object. See “Creating the PortfolioCVaR Object” on page 5-22 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 5-20.

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 main solver algorithm (`'MainSolverOptions'`).
- 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 (for example, 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 main solver (for example, 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 main solver options (`'MainSolverOptions'`), 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 5-78. Specifically, use `estimateBounds` to examine your portfolio set, and use `checkFeasibility` to ensure that your initial portfolio is either feasible and, if infeasible, that you have sufficient turnover to get from your initial portfolio to the portfolio set.

Tip To correct this problem, try solving your problem with larger values for turnover and gradually reduce to the value that you want.

Efficient Portfolios Do Not Make Sense

If you obtain efficient portfolios that, do not seem to make sense, this can happen if you forget to set specific constraints or you set incorrect constraints. For example, if you allow portfolio weights to fall between `0` and `1` and do not set a budget constraint, you can get portfolios that are 100% invested in every asset. Although it may be hard to detect, the best thing to do is to review the constraints you have set with display of the `PortfolioCVaR` object. If you get portfolios with 100% invested in each asset, you can review the display of your object and quickly see that no budget constraint is set. Also, you can use `estimateBounds` and `checkFeasibility` to determine if the bounds for your portfolio set make sense and to determine if the portfolios you obtained are feasible relative to an independent formulation of your portfolio set.

See Also

`PortfolioCVaR` | `estimateScenarioMoments` | `checkFeasibility`

Related Examples

- "Postprocessing Results to Set Up Tradable Portfolios" on page 5-110
- "Creating the PortfolioCVaR Object" on page 5-22
- "Working with CVaR Portfolio Constraints Using Defaults" on page 5-50
- "Troubleshooting for Setting 'Conditional' BoundType, MinNumAssets, and MaxNumAssets Constraints" on page 4-139
- "Asset Returns and Scenarios Using PortfolioCVaR Object" on page 5-36
- "Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object" on page 5-82
- "Estimate Efficient Frontiers for PortfolioCVaR Object" on page 5-102
- "Hedging Using CVaR Portfolio Optimization" on page 5-118
- "Compute Maximum Reward-to-Risk Ratio for CVaR Portfolio" on page 5-130

More About

- "PortfolioCVaR Object" on page 5-17
- "Portfolio Optimization Theory" on page 5-3
- "PortfolioCVaR Object Workflow" on page 5-16

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- CVaR Portfolio Optimization (4 min 56 sec)

Hedging Using CVaR Portfolio Optimization

This example shows how to model two hedging strategies using CVaR portfolio optimization with a PortfolioCVaR object. First, you simulate the price movements of a stock by using a gbm object with simByEuler. Then you use CVaR portfolio optimization to estimate the efficient frontier of the portfolios for the returns at the horizon date. Finally, you compare the CVaR portfolio to a mean-variance portfolio to demonstrate the differences between these two types of risk measures.

Monte Carlo Simulation of Asset Scenarios

Scenarios are required to define and evaluate the CVaR portfolio. These scenarios can be generated in multiple ways, as well as obtained from historical observations. This example uses a Monte Carlo simulation of a geometric Brownian motion to generate the scenarios. The Monte Carlo simulation produces trials of a stock price after one year. These stock prices provide the returns of five different investment strategies that define the assets in the CVaR portfolio. This scenario generation strategy can be further generalized to simulations with more stocks, instruments, and strategies.

Define a stock profile.

```
% Price at time 0
Price_0 = 200;

% Drift (annualized)
Drift = 0.08;

% Volatility (annualized)
Vol = 0.4;

% Valuation date
Valuation = datetime(2012,1,1);

% Investment horizon date
Horizon = datetime(2013,1,1);

% Risk-free rate
RiskFreeRate = 0.03;
```

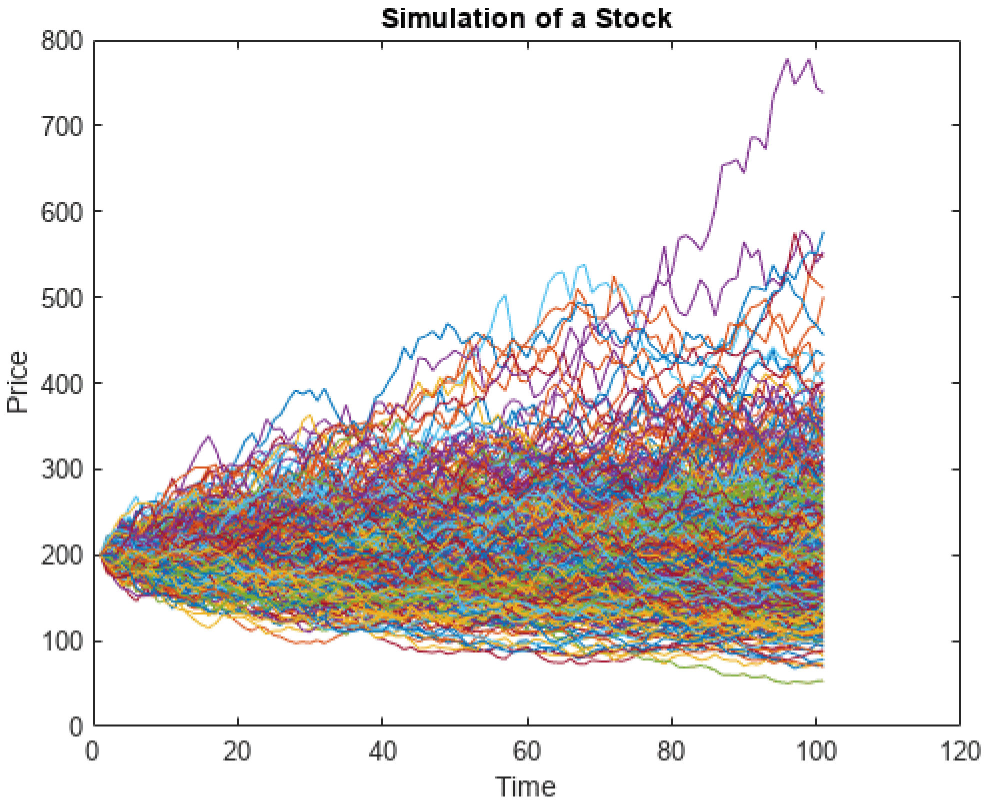
Simulate the price movements of the stock from the valuation date to the horizon date using a gbm object with simByEuler.

```
% Number of trials for the Monte Carlo simulation
NTRIALS = 100000;

% Length (in years) of the simulation
T = date2time(Valuation, Horizon, 1, 1);

% Number of periods per trial (approximately 100 periods per year)
NPERIODS = round(100*T);

% Length (in years) of each time step per period
dt = T/NPERIODS;

% Instantiate the gbm object
StockGBM = gbm(Drift, Vol, 'StartState', Price_0);
```

```
% Run the simulation
Paths = StockGBM.simByEuler(NPERIODS, 'NTRIALS', NTRIALS, ...
    'DeltaTime', dt, 'Antithetic', true);
```

Plot the simulation of the stock. For efficiency, plot only some scenarios.

```
plot(squeeze(Paths(:,:,1:500)));
title('Simulation of a Stock');
xlabel('Time');
ylabel('Price');
```



Calculate the prices of different put options using a Black-Scholes model with the `blsprice` function.

```
% Put option with strike at 50% of current underlying price
Strike50 = 0.50*Price_0;
[~, Put50] = blsprice(Price_0, Strike50, RiskFreeRate, T, Vol);

% Put option with strike at 75% of current underlying price
Strike75 = 0.75*Price_0;
[~, Put75] = blsprice(Price_0, Strike75, RiskFreeRate, T, Vol);

% Put option with strike at 90% of current underlying price
Strike90 = 0.90*Price_0;
[~, Put90] = blsprice(Price_0, Strike90, RiskFreeRate, T, Vol);

% Put option with strike at 95% of current underlying price
Strike95 = 0.95*Price_0; % Same as strike
[~, Put95] = blsprice(Price_0, Strike95, RiskFreeRate, T, Vol);
```

The goal is to find the efficient portfolio frontier for the returns at the horizon date. Hence, obtain the scenarios from the trials of the Monte Carlo simulation at the end of the simulation period.

```
Price_T = squeeze(Paths(end, 1, :));
```

Generate the scenario matrix using five strategies. The first strategy is a stock-only strategy; the rest of the strategies are the stock hedged with put options at different strike price levels. To compute the returns from the prices obtained by the Monte Carlo simulation for the stock-only strategy, divide the change in the stock price `(Price_T - Price_0)` by the initial price `Price_0`. To compute the returns of the stock with different put options, first compute the "observed price" at the horizon date, that is, the stock price with the acquired put option taken into account. If the stock price at the horizon date is less than the strike price, the observed price is the strike price. Otherwise, the observed price is the true stock price. This observed price is represented by the formula

$$\text{Observed price} = \max\{\text{Price}_T, \text{ Strike}\}.$$

You can then compute the "initial cost" of the stock with the put option, which is given by the initial stock price `Price_0` plus the price of the put option. Finally, to compute the return of the put option strategies, divide the observed price minus the initial cost by the initial cost.

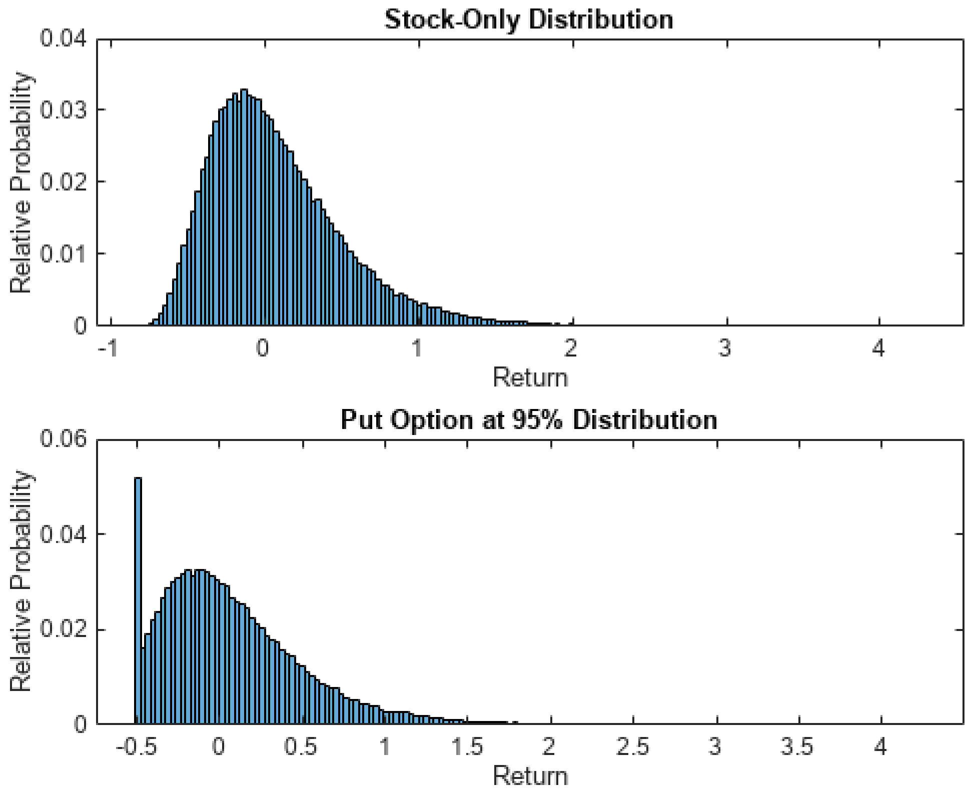
```
AssetScenarios = zeros(NTRIALS, 5);

% Strategy 1: Stock only
AssetScenarios(:, 1) = (Price_T - Price_0) ./ Price_0;

% Strategy 2: Put option cover at 50%
AssetScenarios(:, 2) = (max(Price_T, Strike50) - (Price_0 + Put50)) ./ ...
    (Price_0 + Put50);

% Strategy 2: Put option cover at 75%
AssetScenarios(:, 3) = (max(Price_T, Strike75) - (Price_0 + Put75)) ./ ...
    (Price_0 + Put75);

% Strategy 2: Put option cover at 90%
AssetScenarios(:, 4) = (max(Price_T, Strike90) - (Price_0 + Put90)) ./ ...
    (Price_0 + Put90);

% Strategy 2: Put option cover at 95%
AssetScenarios(:, 5) = (max(Price_T, Strike95) - (Price_0 + Put95)) ./ ...
    (Price_0 + Put95);
```

The portfolio weights associated with each of the five assets previously defined represent the percentage of the total weatlh to invest in each strategy. For example, consider a portfolio with the weights `[0.5 0 0.5 0]`. The weights indicate that the best allocation is to invest 50% in the stock-only strategy and the remaining 50% in a put option at 75%.

Plot the distribution of the returns for the stock-only strategy and the put option at the 95% strategy. Notice that the returns are not normally distributed. For a mean-variance portfolio, a lack of symmetry in the plotted returns usually indicates poor mean-variance portfolio performance since variance, as a risk measure, is not sensitive to skewed distributions.

```
% Create histogram
figure;

% Stock only
subplot(2,1,1);
```

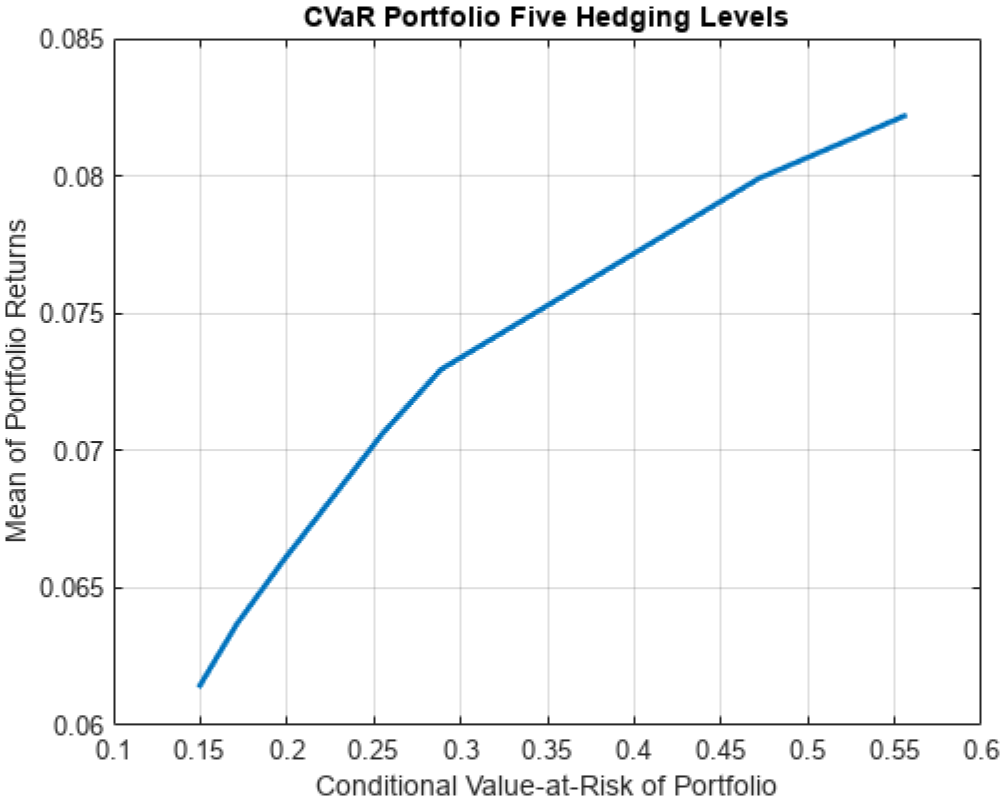
```
histogram(AssetScenarios(:,1),'Normalization','probability')
title('Stock-Only Distribution')
xlabel('Return')
ylabel('Relative Probability')

% Put option cover
subplot(2,1,2);
histogram(AssetScenarios(:,2),'Normalization','probability')
title('Put Option at 95% Distribution')
xlabel('Return')
ylabel('Relative Probability')
```



CVaR Efficient Frontier

Create a `PortfolioCVaR` object using the `AssetScenarios` from the simulation.

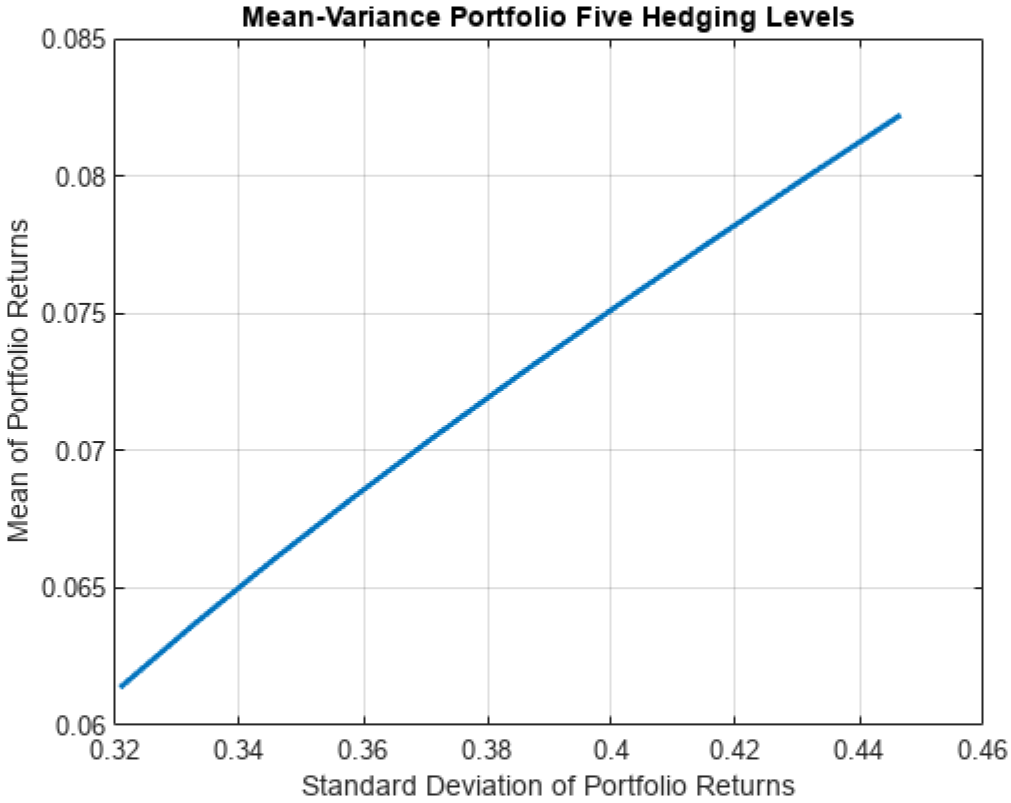
```
p = PortfolioCVaR('Name','CVaR Portfolio Five Hedging Levels',...
    'AssetList',{'Stock','Hedge50','Hedge75','Hedge90','Hedge95'},...
    'Scenarios', AssetScenarios, 'LowerBound', 0, ...
    'Budget', 1, 'ProbabilityLevel', 0.95);

% Estimate the efficient frontier to obtain portfolio weights
pwgt = estimateFrontier(p);

% Plot the efficient frontier
figure;
plotFrontier(p, pwgt);
```



CVaR and Mean-Variance Efficient Frontier Comparison

Create the `Portfolio` object. Use `AssetScenarios` from the simulation to estimate the assets moments. Notice that, unlike for the `PortfolioCVaR` object, only an estimate of the assets moments is required to fully specify a mean-variance portfolio.

```
pmv = Portfolio('Name','Mean-Variance Portfolio Five Hedging Levels',...
   'AssetList',{'Stock','Hedge50','Hedge75','Hedge90','Hedge95'});
pmv = estimateAssetMoments(pmv, AssetScenarios);
pmv = setDefaultConstraints(pmv);

% Estimate the efficient frontier to obtain portfolio weights
pwgtmv = estimateFrontier(pmv);

% Plot the efficient frontier
figure;
plotFrontier(pmv, pwgtmv);
```

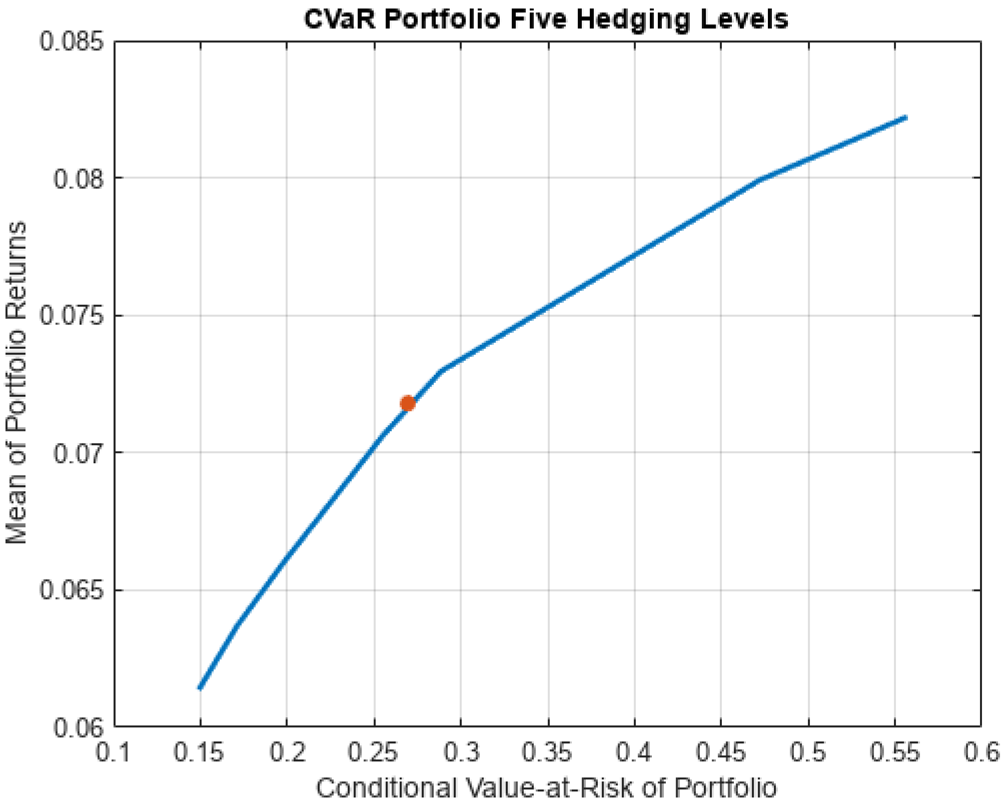


Select a target return. In this case, the target return is the midpoint return on the CVaR portfolio efficient frontier.

```
% Achievable levels of return
pretlimits = estimatePortReturn(p, estimateFrontierLimits(p));
TargetRet = mean(pretlimits); % Target half way up the frontier
```

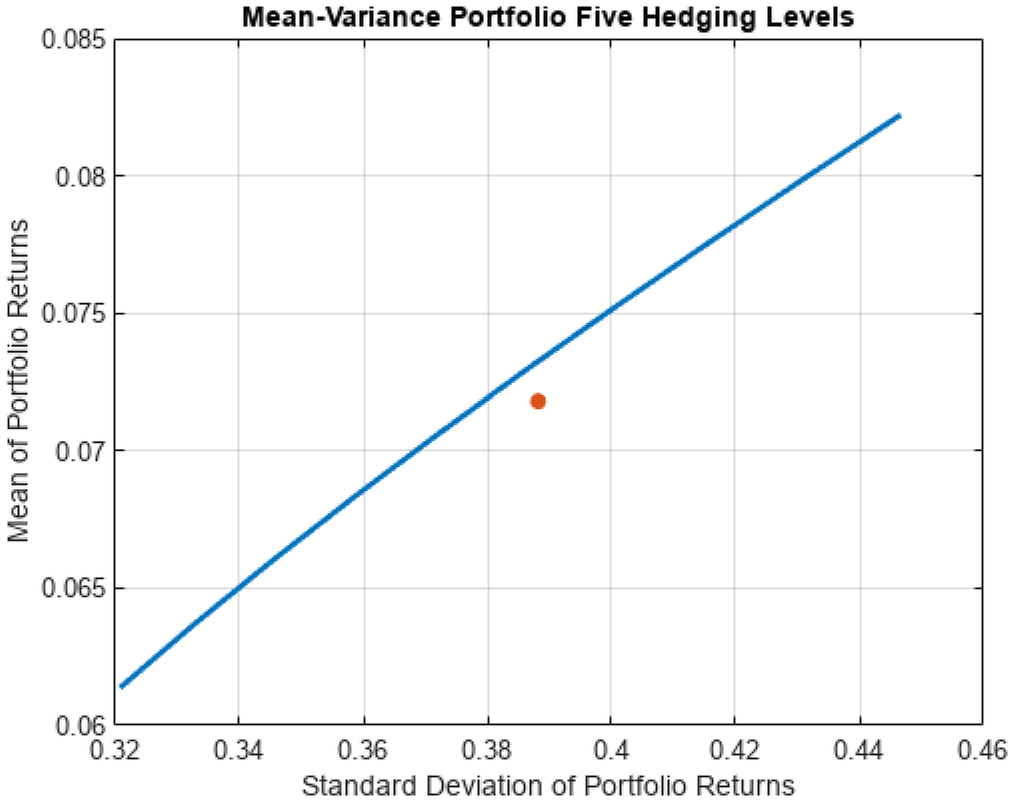
Plot the efficient CVaR portfolio for the return `TargetRet` on the CVaR efficient frontier.

```
% Obtain risk level at target return
pwgtTarget = estimateFrontierByReturn(p,TargetRet); % CVaR efficient portfolio
priskTarget = estimatePortRisk(p,pwgtTarget);

% Plot point onto CVaR efficient frontier
figure;
plotFrontier(p,pwgt);
hold on
scatter(priskTarget,TargetRet,[],'filled');
hold off
```



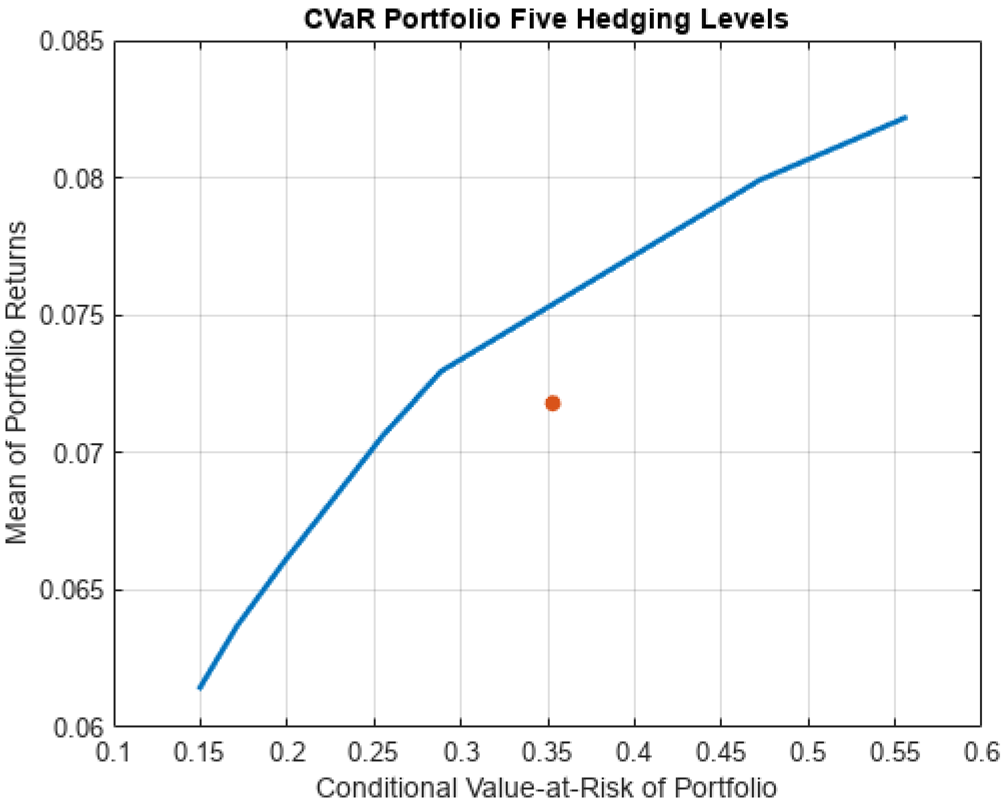
Plot the efficient CVaR portfolio for the return `TargetRet` on the mean-variance efficient frontier. Notice that the efficient CVaR portfolio is below the mean-variance efficient frontier.

```
% Obtain the variance for the efficient CVaR portfolio
pmvretTarget = estimatePortReturn(pmv,pwgtTarget); % Should be TargetRet
pmvriskTarget = estimatePortRisk(pmv,pwgtTarget);  % Risk proxy is variance

% Plot efficient CVaR portfolio onto mean-variance frontier
figure;
plotFrontier(pmv,pwgtmv);
hold on
scatter(pmvriskTarget,pmvretTarget,[],'filled');
hold off
```



Plot the efficient mean-variance portfolio for the return `TargetRet` on the CVaR efficient frontier. Notice that the efficient mean-variance portfolio is below the CVaR efficient frontier.

```
% Obtain the mean-variance efficient portfolio at target return
pwgtmvTarget = estimateFrontierByReturn(pmv,TargetRet);

% Obtain the CVaR risk for the mean-variance efficient portfolio
pretTargetCVaR = estimatePortReturn(p,pwgtmvTarget); % Should be TargetRet
priskTargetCVaR = estimatePortRisk(p,pwgtmvTarget);  % Risk proxy is CVaR

% Plot mean-variance efficient portfolio onto the CVaR frontier
figure;
plotFrontier(p,pwgt);
hold on
scatter(priskTargetCVaR,pretTargetCVaR,[],'filled');
hold off
```



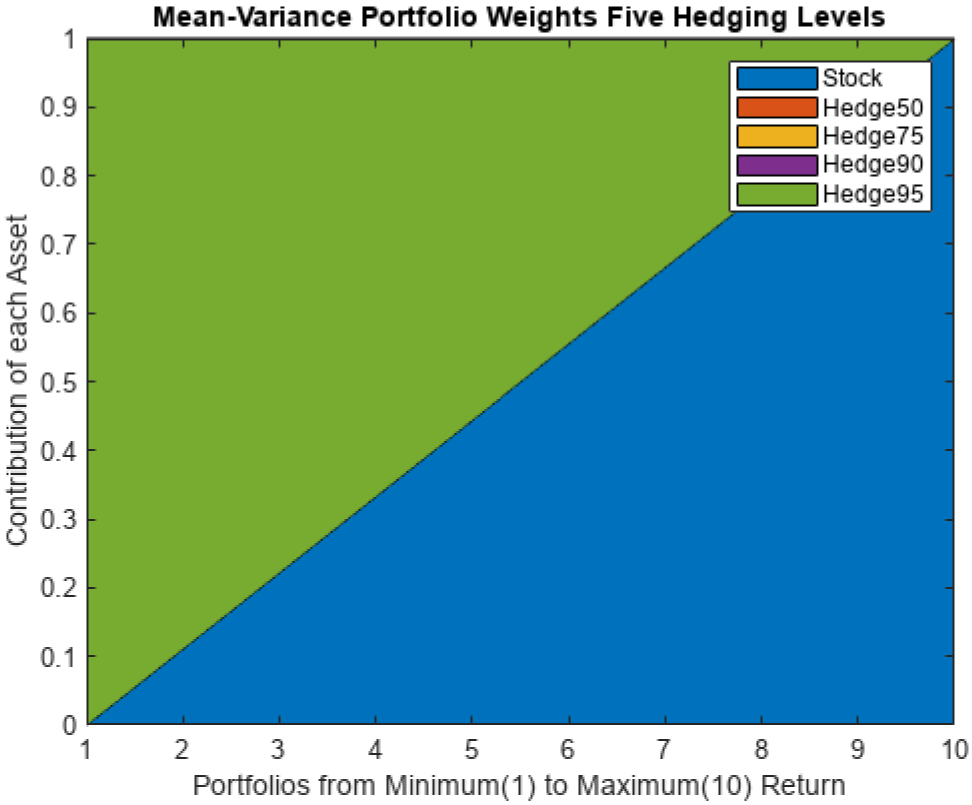
Since mean-variance and CVaR are two different risk measures, this example illustrates that the efficient portfolio for one type of risk measure is not efficient for the other.

CVaR and Mean-Variance Portfolio Weights Comparison

Examine the portfolio weights of the porfolios that make up each efficient frontier to obtain a more detailed comparison between the mean-variance and CVaR efficient frontiers.

Plot the weights associated with the mean-variance portfolio efficient frontier.

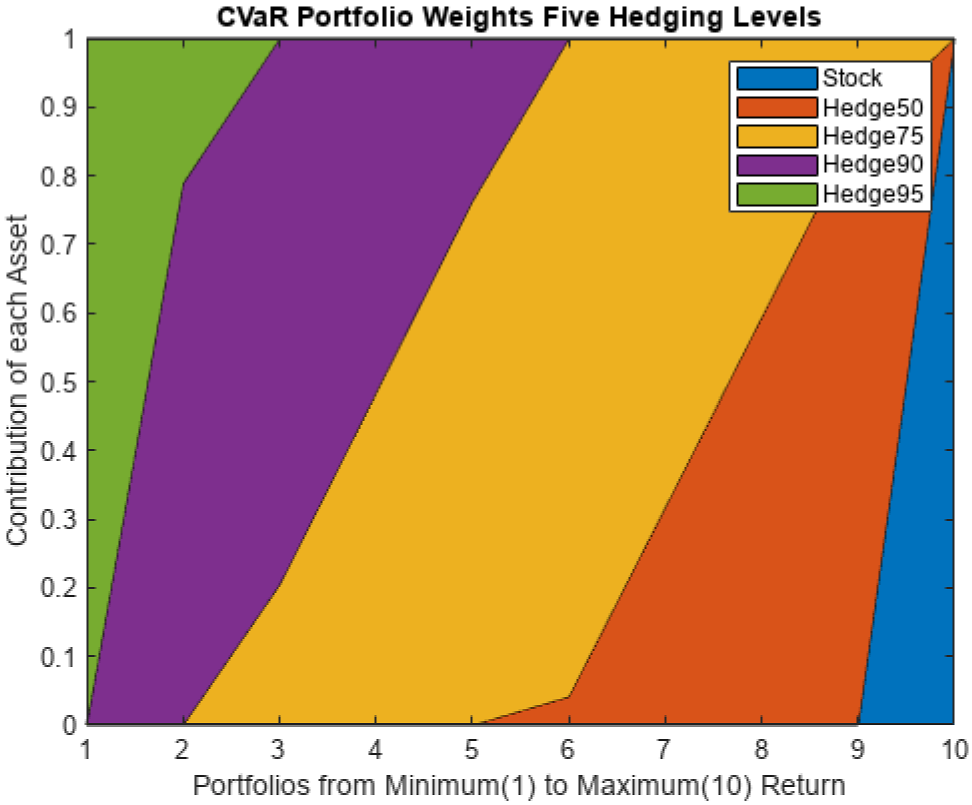
```
% Plot the mean-variance portfolio weights
figure;
area(pwgtmv');
legend(pmv.AssetList);
axis([1 10 0 1])
title('Mean-Variance Portfolio Weights Five Hedging Levels');
xlabel('Portfolios from Minimum(1) to Maximum(10) Return');
ylabel('Contribution of each Asset');
```



The weights associated with the mean-variance portfolios on the efficient frontier use only two strategies, `'Stock'` and `'Hedge95'`. This behavior is an effect of the correlations among the five assets in the portfolio. Because the correlations between the assests are close to 1, the standard deviation of the portfolio is almost a linear combination of the standard deviation of the assets. Hence, because assets with larger returns are associated with assets with larger variance, a linear combination of only the assets with the smallest and largest returns is observed in the efficient frontier.

Plot the weights associated with the CVaR portfolio efficient frontier.

```
% Plot the CVaR portfolio weights
figure;
area(pwgt');
legend(p.AssetList);
axis([1 10 0 1])
title('CVaR Portfolio Weights Five Hedging Levels');
xlabel('Portfolios from Minimum(1) to Maximum(10) Return');
ylabel('Contribution of each Asset');
```



For both types of portfolios, mean-variance and CVaR, the portfolio with the maximum expected return is the one that allocates all the weight to the stock-only strategy. This makes sense because no put options are acquired, which translates into larger returns. Also, the portfolio with minimum variance is the same for both risk measures. This portfolio is the one that allocates everything to `'Hedge95'` because that strategy limits the possible losses the most. The real differences between the two types of porfolios are observed for the return levels between the minimum and maximum. There, in contrast to the efficient portfolios obtained using variance as the measure of risk, the weights of the CVaR portfolios range among all five possible strategies.

See Also

`PortfolioCVaR` | `getScenarios` | `setScenarios` | `estimateScenarioMoments` | `simulateNormalScenariosByMoments` | `simulateNormalScenariosByData` | `setCosts` | `checkFeasibility`

Related Examples

- “Troubleshooting CVaR Portfolio Optimization Results” on page 5-115
- “Creating the PortfolioCVaR Object” on page 5-22
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-50
- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-36
- “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-82
- “Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-102

- “Compute Maximum Reward-to-Risk Ratio for CVaR Portfolio” on page 5-130

More About

- “PortfolioCVaR Object” on page 5-17
- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-16

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- CVaR Portfolio Optimization (4 min 56 sec)

Compute Maximum Reward-to-Risk Ratio for CVaR Portfolio

Create a `PortfolioCVaR` object and incorporate a list of assets from `CAPMUniverse.mat`. Use `simulateNormalScenariosByData` to simulate the scenarios for each of the assets. These portfolio constraints require fully invested long-only portfolios (nonnegative weights that must sum to 1).

```
rng(1) % Set the seed for reproducibility.
load CAPMuniverse

p = PortfolioCVaR('AssetList',Assets(1:12));
p = simulateNormalScenariosByData(p, Data(:,1:12), 20000 ,'missingdata',true);
p = setProbabilityLevel(p, 0.95);
p = setDefaultConstraints(p);
disp(p)
```

```
  PortfolioCVaR with properties:

             BuyCost: []
            SellCost: []
        RiskFreeRate: []
    ProbabilityLevel: 0.9500
            Turnover: []
         BuyTurnover: []
        SellTurnover: []
        NumScenarios: 20000
                Name: []
           NumAssets: 12
           AssetList: {'AAPL'  'AMZN'  'CSCO'  'DELL'  'EBAY'  'GOOG'  'HPQ'  'IBM'  'INTC'  'MS
            InitPort: []
         AInequality: []
         bInequality: []
           AEquality: []
           bEquality: []
          LowerBound: [12x1 double]
          UpperBound: []
         LowerBudget: 1
         UpperBudget: 1
         GroupMatrix: []
          LowerGroup: []
          UpperGroup: []
              GroupA: []
              GroupB: []
          LowerRatio: []
          UpperRatio: []
        MinNumAssets: []
        MaxNumAssets: []
           BoundType: [12x1 categorical]
```

To obtain the portfolio that maximizes the reward-to-risk ratio (which is equivalent to the Sharpe ratio for mean-variance portfolios), search on the efficient frontier iteratively for the porfolio that minimizes the negative of the reward-to-risk ratio:

$$-\frac{\text{portfolio return} - \text{risk free rate}}{\text{portfolio CVaR}}.$$

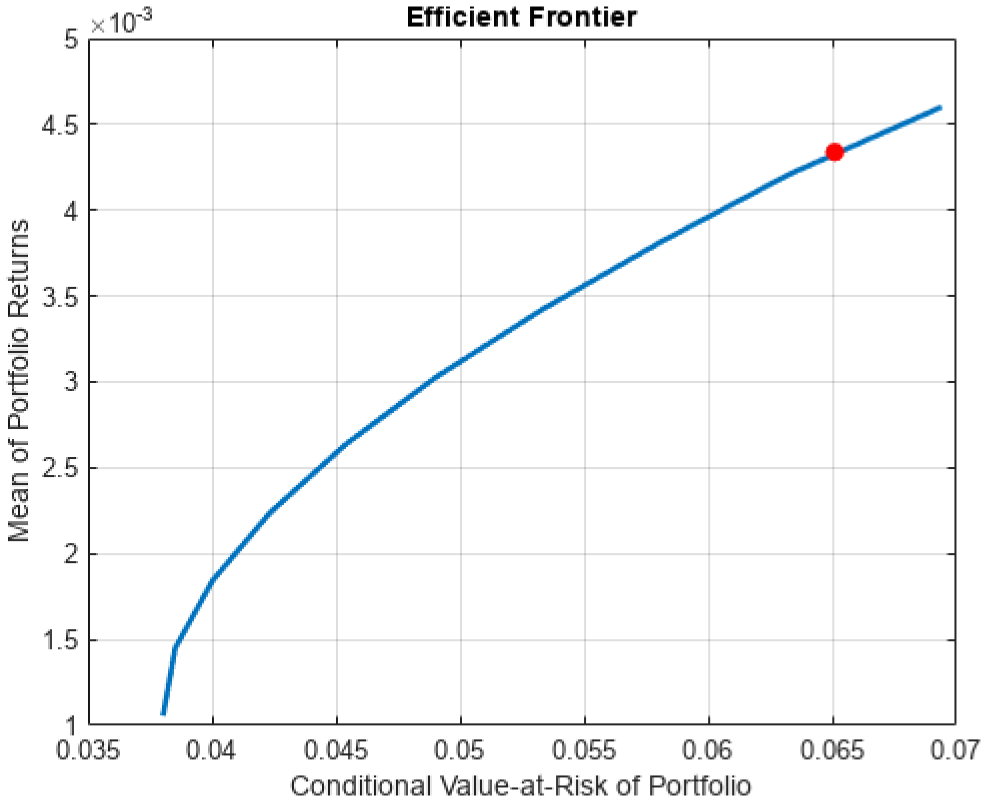
To do so, use the `sratio` function, defined in the Local Functions on page 5-132 section, to return the negative reward-to-risk ratio for a target return. Then, pass this function to `fminbnd`. `fminbnd` iterates through the possible return values and evaluates their associated reward-to-risk ratio. `fminbnd` returns the optimal return for which the maximum reward-to-risk ratio is achieved (or that minimizes the negative of the reward-to-risk ratio).

```
% Obtain the minimum and maximum returns of the portfolio.
pwgtLimits = estimateFrontierLimits(p);
retLimits = estimatePortReturn(p,pwgtLimits);
minret = retLimits(1);
maxret = retLimits(2);

% Search on the frontier iteratively. Find the return that minimizes the
% negative of the reward-to-risk ratio.
fhandle = @(ret) iterative_local_obj(ret,p);
options = optimset('Display', 'off', 'TolX', 1.0e-8);
optret = fminbnd(fhandle, minret, maxret, options);

% Obtain the portfolio weights associated with the return that achieves
% the maximum reward-to-risk ratio.
pwgt = estimateFrontierByReturn(p,optret)
```

```
pwgt = 12×1

    0.0885
         0
         0
         0
         0
    0.9115
         0
         0
         0
         0
      ⋮
```

Use `plotFrontier` to plot the efficient frontier and `estimatePortRisk` to estimate the maximum reward-to-risk ratio portfolio.

```
plotFrontier(p);
hold on
% Compute the risk level for the maximum reward-to-risk ratio portfolio.
optrsk = estimatePortRisk(p,pwgt);
scatter(optrsk,optret,50,'red','filled')
hold off
```



Local Functions

This local function that computes the negative of the reward-to-risk ratio for a target return level.

```
function sratio = iterative_local_obj(ret, obj)
% Set the objective function to the negative of the reward-to-risk ratio.

risk = estimatePortRisk(obj,estimateFrontierByReturn(obj,ret));

if ~isempty(obj.RiskFreeRate)
    sratio = -(ret - obj.RiskFreeRate)/risk;
else
    sratio = -ret/risk;
end

end
```

See Also

PortfolioCVaR | getScenarios | setScenarios | estimateScenarioMoments | simulateNormalScenariosByMoments | simulateNormalScenariosByData | setCosts | checkFeasibility

Related Examples

- “Troubleshooting CVaR Portfolio Optimization Results” on page 5-115

- “Creating the PortfolioCVaR Object” on page 5-22
- “Working with CVaR Portfolio Constraints Using Defaults” on page 5-50
- “Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-36
- “Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-82
- “Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-102
- “Hedging Using CVaR Portfolio Optimization” on page 5-118

More About

- “PortfolioCVaR Object” on page 5-17
- “Portfolio Optimization Theory” on page 5-3
- “PortfolioCVaR Object Workflow” on page 5-16

External Websites

- Getting Started with Portfolio Optimization (4 min 13 sec)
- CVaR Portfolio Optimization (4 min 56 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-5) 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 supports mean-variance portfolio optimization (see Markowitz [46], [47] at “Portfolio Optimization” on page A-5). 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 implements what is known as conditional value-at-risk portfolio optimization (see Rockafellar and Uryasev [48], [49] at “Portfolio Optimization” on page A-5), 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 implements what is known as mean-absolute deviation portfolio optimization (see Konno and Yamazaki [50] at “Portfolio Optimization” on page A-5), which is generally referred to as MAD portfolio optimization. MAD portfolio optimization works with the same return proxies and portfolio sets as mean-variance portfolio optimization but uses mean-absolute deviation portfolio returns as the risk proxy.

Return Proxy

The proxy for portfolio return is a function $\mu : X \rightarrow R$ on a portfolio set $X \subset R^n$ that characterizes the rewards associated with portfolio choices. 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 $y_1,...,y_S$ 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

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

where:

r_0 is the risk-free rate (scalar).

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

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

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

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

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

Risk Proxy

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

Variance

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

$$Variance(x) = x^T Cx$$

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 [46], [47] at (“Portfolio Optimization” on page A-5).

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 –α). 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

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

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-5.

Mean Absolute-Deviation

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

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

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-5.

See Also

`Portfolio` | `PortfolioCVaR` | `PortfolioMAD`

Related Examples

- “Creating the PortfolioMAD Object” on page 6-21
- “Working with MAD Portfolio Constraints Using Defaults” on page 6-48

More About

- “PortfolioMAD Object” on page 6-16
- “Portfolio Set for Optimization Using PortfolioMAD Object” on page 6-7
- “Default Portfolio Problem” on page 6-14
- “PortfolioMAD Object Workflow” on page 6-15

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 and which are properties for the `PortfolioMAD` object:

- Linear inequality constraints
- Linear equality constraints
- `'Simple'` Bound constraints
- `'Conditional'` Bond constraints
- Budget constraints
- Group constraints
- Group ratio constraints
- Average turnover constraints
- One-way turnover constraints
- Cardinality constraints

Linear Inequality Constraints

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

$$A_I x \leq b_I$$

where:

x is the portfolio (n vector).

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

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

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

`PortfolioMAD` 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.

`PortfolioMAD` 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.

'Simple' Bound Constraints

`'Simple'` 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

$$l_B \leq x \leq u_B$$

where:

x is the portfolio (n vector).

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

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

n is the number of assets in the universe.

`PortfolioMAD` 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 6-14) has l_B = `0` with u_B set implicitly through a budget constraint.

'Conditional' Bound Constraints

`'Conditional'` Bound constraints, also called semicontinuous constraints, are mixed-integer linear constraints that confine portfolio weights to fall either above or below specific bounds *if* the asset is selected; otherwise, the value of the asset is zero. Use `setBounds` to specify bound constraints with a `'Conditional'` `BoundType`. To mathematically formulate this type of constraints, a binary variable v_i is needed. $v_i = 0$ indicates that asset i is not selected and v_i indicates that the asset was selected. Thus

$$l_i v_i \leq x_i \leq u_i v_i$$

where

x is the portfolio (n vector).

l is the conditional lower-bound constraint (n vector).

u is the conditional upper-bound constraint (n vector).

n is the number of assets in the universe.

`PortfolioMAD` object properties to specify the bound constraint are:

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

The default is to ignore this constraint.

Budget Constraints

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

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

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.

`PortfolioMAD` object properties to specify budget constraints are:

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

The default is to ignore this constraint.

The default portfolio optimization problem (see “Default Portfolio Problem” on page 6-14) has $l_S = u_S = 1$, which means that the portfolio weights sum to 1. If the portfolio optimization problem includes possible movements in and out of cash, the budget constraint specifies how far portfolios can go into cash. For example, if $l_S = 0$ and $u_S = 1$, then the portfolio can have 0–100% invested in cash. If cash is to be a portfolio choice, set RiskFreeRate (r_0) to a suitable value (see “Return Proxy” on page 6-3 and “Working with a Riskless Asset” on page 6-43).

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

PortfolioMAD 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

$$l_{Ri}(G_Bx)_i \le (G_Ax)_i \le u_{Ri}(G_Bx)_i$$

for $i = 1,\ldots, n_R$ 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.

`PortfolioMAD` 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 compute portfolio turnover as the average of purchases and sales. Average turnover constraints take the form

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

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.

`PortfolioMAD` 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

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

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

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 `PortfolioMAD` 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 6-70) with τ is not a combination of the one-way turnover constraints with $\tau = \tau_B = \tau_S$.

Cardinality Constraints

Cardinality constraint limits the number of assets in the optimal allocation for an `PortfolioMAD` object. Use `setMinMaxNumAssets` to specify the `'MinNumAssets'` and `'MaxNumAssets'` constraints. To mathematically formulate this type of constraints, a binary variable v_i is needed. $v_i = 0$ indicates that asset i is not selected and $v_i = 1$ indicates that the asset was selected. Thus

$$MinNumAssets \leq \sum_{i=1}^{NumAssets} v_i \leq MaxNumAssets$$

The default is to ignore this constraint.

See Also

`Portfolio` | `PortfolioCVaR` | `PortfolioMAD`

Related Examples

- "Creating the PortfolioMAD Object" on page 6-21
- "Working with MAD Portfolio Constraints Using Defaults" on page 6-48

More About

- "PortfolioMAD Object" on page 6-16
- "Default Portfolio Problem" on page 6-14
- "PortfolioMAD Object Workflow" on page 6-15

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 6-21
- “Working with MAD Portfolio Constraints Using Defaults” on page 6-48

More About

- “PortfolioMAD Object” on page 6-16
- “Portfolio Set for Optimization Using PortfolioMAD Object” on page 6-7
- “PortfolioMAD Object Workflow” on page 6-15

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. For more information, see “Creating the PortfolioMAD Object” on page 6-21.
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 6-34.
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, `'Conditional'` `BoundType`, and `MinNumAssets`, `MaxNumAssets` constraints. For more information, see “Working with MAD Portfolio Constraints Using Defaults” on page 6-48 and “Working with 'Conditional' BoundType, MinNumAssets, and MaxNumAssets Constraints Using PortfolioMAD Objects” on page 6-67.
4. Validate the MAD Portfolio.

   Identify errors for the portfolio specification. For more information, see “Validate the MAD Portfolio Problem” on page 6-76.
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 6-80 and “Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-97.
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 6-105.

See Also

More About

- “Portfolio Optimization Theory” on page 6-2

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 `PortfolioMAD` 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 `PortfolioMAD` to set the properties for the `PortfolioMAD` object.

`PortfolioMAD` 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 6-21.

Setting and Getting Properties

You can set properties of a PortfolioMAD object using either PortfolioMAD 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 object 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 object, 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 object and set functions have several useful features:

- The PortfolioMAD object and set functions try to determine the dimensions of your problem with either explicit or implicit inputs.
- The PortfolioMAD object and set functions try to resolve ambiguities with default choices.
- The PortfolioMAD object 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. An efficient portfolio are the portfolios that satisfy the criteria of minimum risk for a given level of return and maximum return for a given level of risk. 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 enable 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)
```

```
3×2 PortfolioMAD array 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
    MinNumAssets
    MaxNumAssets
    BoundType
```

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 `PortfolioMAD` 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 6-21
- “Working with MAD Portfolio Constraints Using Defaults” on page 6-48

More About

- “Portfolio Optimization Theory” on page 6-2

- “PortfolioMAD Object Workflow” on page 6-15

Creating the PortfolioMAD Object

In this section...

To create a fully specified MAD portfolio optimization problem, instantiate the `PortfolioMAD` object using `PortfolioMAD`. For information on the workflow when using `PortfolioMAD` objects, see "PortfolioMAD Object Workflow" on page 6-15.

Syntax

Use `PortfolioMAD` to create an instance of an object of the `PortfolioMAD` class. You can use the `PortfolioMAD` object 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` object also accepts collections of argument name-value pair arguments for properties and their values. The `PortfolioMAD` object 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 `PortfolioMAD` 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 6-25). The `PortfolioMAD` object 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: []
    MinNumAssets: []
    MaxNumAssets: []
       BoundType: []
```

The approaches listed provide a way to set up a portfolio optimization problem with the `PortfolioMAD` object. 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` object 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:

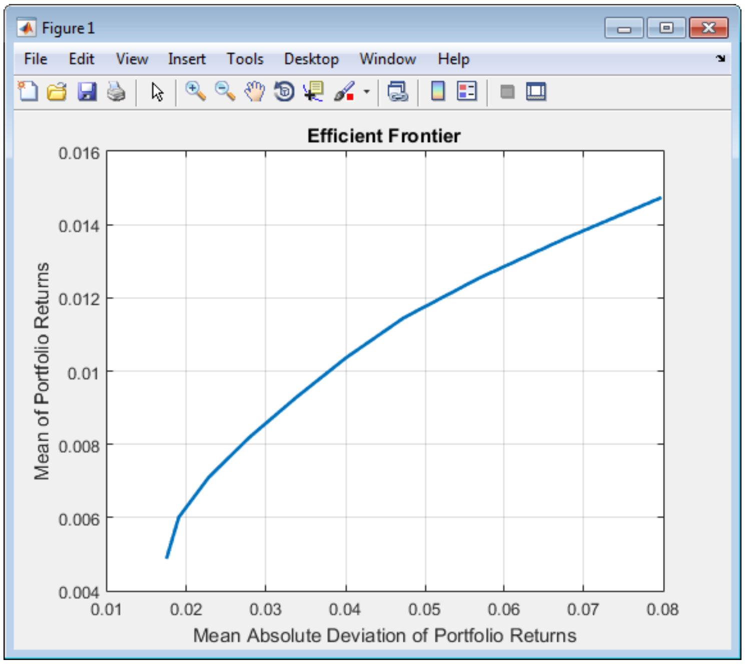
```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

AssetScenarios = mvnrnd(m, C, 20000);

p = PortfolioMAD('Scenarios', AssetScenarios, ...
'LowerBound', 0, 'LowerBudget', 1, 'UpperBudget', 1)
```

```
p = 

  PortfolioMAD with properties:

         BuyCost: []
        SellCost: []
    RiskFreeRate: []
        Turnover: []
     BuyTurnover: []
    SellTurnover: []
    NumScenarios: 20000
            Name: []
       NumAssets: 4
       AssetList: []
        InitPort: []
     AInequality: []
     bInequality: []
       AEquality: []
       bEquality: []
      LowerBound: [4×1 double]
      UpperBound: []
     LowerBudget: 1
     UpperBudget: 1
     GroupMatrix: []
      LowerGroup: []
      UpperGroup: []
          GroupA: []
          GroupB: []
      LowerRatio: []
      UpperRatio: []
    MinNumAssets: []
    MaxNumAssets: []
       BoundType: []
```

The LowerBound property value undergoes scalar expansion since AssetScenarios provides the dimensions of the problem.

You can use dot notation with the function plotFrontier.

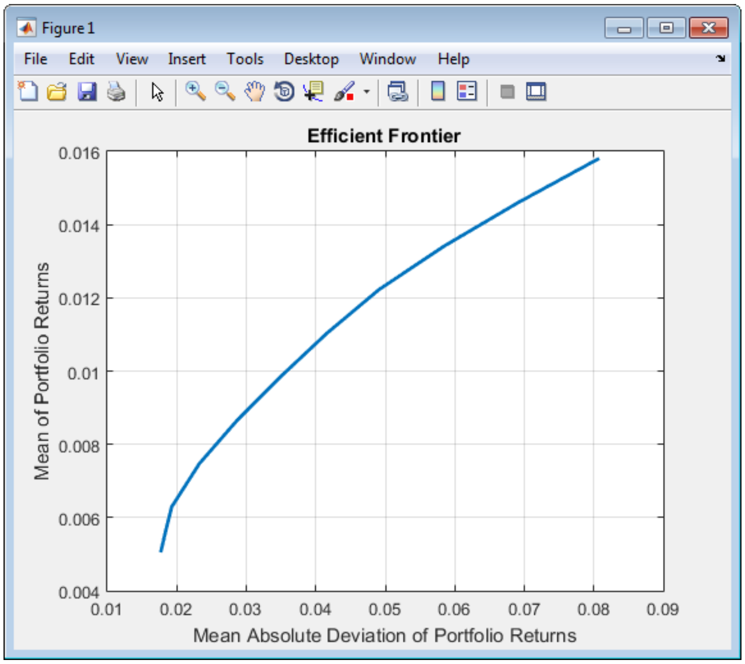
```
p.plotFrontier
```



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` object 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` object 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` object 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` object 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` object. 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 PortfolioMAD 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 using dot notation to a `PortfolioMAD` object. Scenarios must always be set through either the `PortfolioMAD` object, the `setScenarios` function, or any of the scenario simulation functions.

See Also

`PortfolioMAD` | `estimateBounds`

Related Examples

- "Common Operations on the PortfolioMAD Object" on page 6-28
- "Working with MAD Portfolio Constraints Using Defaults" on page 6-48

More About

- "PortfolioMAD Object" on page 6-16
- "Portfolio Optimization Theory" on page 6-2
- "PortfolioMAD Object Workflow" on page 6-15

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 `PortfolioMAD`:

```
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 `PortfolioMAD` 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 object, 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 object:

```
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 | setAssetList | setInitPort | estimateBounds | checkFeasibility

Related Examples

- "Setting Up an Initial or Current Portfolio" on page 6-32
- "Working with MAD Portfolio Constraints Using Defaults" on page 6-48
- "Asset Returns and Scenarios Using PortfolioMAD Object" on page 6-34

More About

- “PortfolioMAD Object” on page 6-16
- “Portfolio Optimization Theory” on page 6-2
- “PortfolioMAD Object Workflow” on page 6-15

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 thecheckFeasibility function described in “Validating MAD Portfolios” on page 6-77. Suppose that you have an initial portfolio in x0, then use the PortfolioMAD object 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 object 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 6-70, “Working with One-Way Turnover Constraints Using PortfolioMAD Object” on page 6-73, and “Working with Transaction Costs” on page 6-44. 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` object 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` | `setAssetList` | `setInitPort` | `estimateBounds` | `checkFeasibility`

Related Examples

- “Common Operations on the PortfolioMAD Object” on page 6-28
- “Working with MAD Portfolio Constraints Using Defaults” on page 6-48
- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-34

More About

- “PortfolioMAD Object” on page 6-16
- “Portfolio Optimization Theory” on page 6-2
- “PortfolioMAD Object Workflow” on page 6-15

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 6-4. 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-5). 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-5) 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 6-15.

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 object 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 object 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 ≤ `startrow` ≤ `endrow` ≤ `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 table or timetable (see “Simulating Normal Scenarios from Time Series Data” on page 6-39). 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 `q`:

```
p = PortfolioMAD;
p = simulateNormalScenariosByData(p, Y, 20000, 'dataformat', 'prices');

[passetmean, passetcovar] = estimateScenarioMoments(p)
```

```
passetmean =

    0.0043
    0.0083
    0.0099
    0.1500


passetcovar =

    0.0053    0.0003    0.0001    0.0002
    0.0003    0.0024    0.0017    0.0010
    0.0001    0.0017    0.0047    0.0027
    0.0002    0.0010    0.0027    0.0100
```

Simulating Normal Scenarios with Missing Data

Often when working with multiple assets, you have missing data indicated by `NaN` values in your return or price data. Although “Multivariate Normal Regression” on page 9-2 goes into detail about regression with missing data, the `simulateNormalScenariosByData` 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 accepts asset returns or prices stored in `table` or `timetable`. The `simulateNormalScenariosByData` function implicitly works with matrices of data or data in a table or timetable object using the same rules for whether the data are returns or prices. To illustrate, use `array2timetable` to create a timetable for 14 assets from `CAPMuniverse` and the use the timetable to simulate scenarios for PortfolioCVaR.

```
load CAPMuniverse
time = datetime(Dates,'ConvertFrom','datenum');
stockTT = array2timetable(Data,'RowTimes',time, 'VariableNames', Assets);
stockTT.Properties
% Notice that GOOG has missing data, because it was not listed before Aug 2004
head(stockTT, 5)
```

```
ans = 

  TimetableProperties with properties:

             Description: ''
                UserData: []
          DimensionNames: {'Time'  'Variables'}
           VariableNames: {'AAPL'  'AMZN'  'CSCO'  'DELL'  'EBAY'  'GOOG'  'HPQ'  'IBM'  'INTC'
    VariableDescriptions: {}
           VariableUnits: {}
      VariableContinuity: []
                RowTimes: [1471×1 datetime]
               StartTime: 03-Jan-2000
              SampleRate: NaN
                TimeStep: NaN
        CustomProperties: No custom properties are set.
      Use addprop and rmprop to modify CustomProperties.


ans =

  5×14 timetable

       Time          AAPL         AMZN         CSCO         DELL         EBAY       GOOG       HP
    ___________    _________    _________    _________    _________    _________    ____    _______

    03-Jan-2000     0.088805       0.1742     0.008775    -0.002353      0.12829    NaN        0.0
    04-Jan-2000    -0.084331     -0.08324     -0.05608     -0.08353    -0.093805    NaN      -0.07
    05-Jan-2000     0.014634     -0.14877    -0.003039     0.070984     0.066875    NaN      -0.00
    06-Jan-2000    -0.086538    -0.060072    -0.016619    -0.038847    -0.012302    NaN      -0.06
    07-Jan-2000     0.047368     0.061013       0.0587    -0.037708    -0.000964    NaN       0.02
```

Use the `'MissingData'` option offered by PortfolioMAD to account for the missing data.

```
p = PortfolioMAD;
p = simulateNormalScenariosByData(p, stockTT, 20000 ,'missingdata',true);
[passetmean, passetcovar] = estimateScenarioMoments(p)
```

```
passetmean = 

    0.0017
    0.0013
    0.0005
    0.0001
    0.0019
    0.0049
    0.0003
    0.0003
    0.0006
   -0.0001
```

```
    0.0005
    0.0011
    0.0002
    0.0001


passetcovar =

    0.0014    0.0005    0.0006    0.0006    0.0006    0.0003    0.0005    0.0003    0.0006    0.0
    0.0005    0.0025    0.0007    0.0005    0.0010    0.0005    0.0005    0.0003    0.0006    0.0
    0.0006    0.0007    0.0013    0.0006    0.0007    0.0004    0.0006    0.0004    0.0008    0.0
    0.0006    0.0005    0.0006    0.0009    0.0006    0.0002    0.0005    0.0003    0.0006    0.0
    0.0006    0.0010    0.0007    0.0006    0.0018    0.0007    0.0005    0.0003    0.0006    0.0
    0.0003    0.0005    0.0004    0.0002    0.0007    0.0013    0.0002    0.0002    0.0002    0.0
    0.0005    0.0005    0.0006    0.0005    0.0005    0.0002    0.0010    0.0003    0.0005    0.0
    0.0003    0.0003    0.0004    0.0003    0.0003    0.0002    0.0003    0.0005    0.0004    0.0
    0.0006    0.0006    0.0008    0.0006    0.0006    0.0002    0.0005    0.0004    0.0011    0.0
    0.0004    0.0004    0.0005    0.0004    0.0005    0.0002    0.0003    0.0002    0.0005    0.0
    0.0005    0.0006    0.0008    0.0005    0.0007    0.0003    0.0005    0.0004    0.0007    0.0
    0.0007    0.0012    0.0008    0.0006    0.0011    0.0011    0.0006    0.0004    0.0007    0.0
    0.0002    0.0002    0.0002    0.0002    0.0002    0.0001    0.0002    0.0002    0.0002    0.0
   -0.0000   -0.0000   -0.0000   -0.0000    0.0000   -0.0000   -0.0000   -0.0000   -0.0000   -0.0
```

Use the name-value input `'DataFormat'` to handle return or price data and `'MissingData'` to ignore or use samples with missing values. In addition, `simulateNormalScenariosByData` extracts asset names or identifiers from a `table` or `timetable` if the argument `'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 `p` from the previous example with the `'GetAssetList'` flag set to `true` extracts the column names from the timetable object:

```
p = simulateNormalScenariosByData(p, stockTT, 20000 ,'missingdata',true, 'GetAssetList', true);
disp(p.AssetList)
```

```
  'AAPL'    'AMZN'    'CSCO'    'DELL'    'EBAY'    'GOOG'    'HPQ'    'IBM'    'INTC'    'MSFT'
```

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 5-29.

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 | simulateNormalScenariosByMoments | simulateNormalScenariosByData

Related Examples

- “Working with a Riskless Asset” on page 6-43
- “Working with Transaction Costs” on page 6-44
- “Creating the PortfolioMAD Object” on page 6-21
- “Working with MAD Portfolio Constraints Using Defaults” on page 6-48
- “Validate the MAD Portfolio Problem” on page 6-76
- “Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object” on page 6-80
- “Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-97

More About

- “PortfolioMAD Object” on page 6-16
- “Portfolio Optimization Theory” on page 6-2
- “PortfolioMAD Object Workflow” on page 6-15

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` object:

```
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` | `simulateNormalScenariosByMoments` | `simulateNormalScenariosByData`

Related Examples

- “Working with Transaction Costs” on page 6-44
- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-34
- “Creating the PortfolioMAD Object” on page 6-21
- “Working with MAD Portfolio Constraints Using Defaults” on page 6-48
- “Validate the MAD Portfolio Problem” on page 6-76
- “Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object” on page 6-80
- “Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-97
- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-34

More About

- “PortfolioMAD Object” on page 6-16
- “Portfolio Optimization Theory” on page 6-2
- “PortfolioMAD Object Workflow” on page 6-15

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 6-3). 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 object. 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` object 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` object 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` | `simulateNormalScenariosByMoments` | `simulateNormalScenariosByData`

Related Examples

- “Working with a Riskless Asset” on page 6-43
- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-34
- “Creating the PortfolioMAD Object” on page 6-21
- “Working with MAD Portfolio Constraints Using Defaults” on page 6-48
- “Validate the MAD Portfolio Problem” on page 6-76
- “Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object” on page 6-80

- “Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-97
- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-34

More About

- “PortfolioMAD Object” on page 6-16
- “Portfolio Optimization Theory” on page 6-2
- “PortfolioMAD Object Workflow” on page 6-15

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 6-15.

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 object 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: [20×1 double]
      UpperBound: []
     LowerBudget: 1
     UpperBudget: 1
     GroupMatrix: []
      LowerGroup: []
      UpperGroup: []
          GroupA: []
          GroupB: []
      LowerRatio: []
      UpperRatio: []
    MinNumAssets: []
    MaxNumAssets: []
       BoundType: []
```

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: [20×1 double]
      UpperBound: []
     LowerBudget: 1
     UpperBudget: 1
     GroupMatrix: []
      LowerGroup: []
      UpperGroup: []
          GroupA: []
          GroupB: []
      LowerRatio: []
      UpperRatio: []
    MinNumAssets: []
    MaxNumAssets: []
       BoundType: [20×1 categorical]
```

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: [20×1 double]
       UpperBound: []
      LowerBudget: 1
      UpperBudget: 1
      GroupMatrix: []
       LowerGroup: []
       UpperGroup: []
           GroupA: []
           GroupB: []
       LowerRatio: []
       UpperRatio: []
     MinNumAssets: []
     MaxNumAssets: []
        BoundType: [20×1 categorical]
```

See Also

PortfolioMAD | setDefaultConstraints | setBounds | setBudget | setGroups | setGroupRatio | setEquality | setInequality | setTurnover | setOneWayTurnover

Related Examples

- “Working with 'Simple' Bound Constraints Using PortfolioMAD Object” on page 6-52
- “Working with Budget Constraints Using PortfolioMAD Object” on page 6-55
- “Working with Group Constraints Using PortfolioMAD Object” on page 6-57
- “Working with Group Ratio Constraints Using PortfolioMAD Object” on page 6-60
- “Working with Linear Equality Constraints Using PortfolioMAD Object” on page 6-63
- “Working with Linear Inequality Constraints Using PortfolioMAD Object” on page 6-65
- “Working with Average Turnover Constraints Using PortfolioMAD Object” on page 6-70
- “Working with One-Way Turnover Constraints Using PortfolioMAD Object” on page 6-73
- “Creating the PortfolioMAD Object” on page 6-21
- “Validate the MAD Portfolio Problem” on page 6-76
- “Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object” on page 6-80
- “Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-97

- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-34

More About

- “PortfolioMAD Object” on page 6-16
- “Portfolio Optimization Theory” on page 6-2
- “PortfolioMAD Object Workflow” on page 6-15

Working with 'Simple' Bound Constraints Using PortfolioMAD Object

'Simple' bound constraints are optional linear constraints that maintain upper and lower bounds on portfolio weights (see “'Simple' Bound Constraints” on page 6-8). 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 4-57).

Setting 'Simple' Bounds Using the PortfolioMAD Function

The properties for bound constraints are set through the PortfolioMAD object. 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, 'BoundType', 'Simple');
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 4-64.

Setting 'Simple' 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,'BoundType', 'Simple');
disp(p.NumAssets)
disp(p.LowerBound)
disp(p.UpperBound)
```

```
  2

  0.5000
```

```
   0.2500

   0.7500
   0.5000
```

Setting 'Simple' Bounds Using the PortfolioMAD Function or setBounds Function

Both the PortfolioMAD object 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,'BoundType', 'Simple');
```

or

```
p = PortfolioMAD('NumAssets', 500);
p = setBounds(p, 0, 0.05,'BoundType','Simple');
```

or

```
p = PortfolioMAD;
p = setBounds(p, 0, 0.05,'NumAssets', 500,'BoundType','Simple');
```

To clear bound constraints from your PortfolioMAD object, use either the PortfolioMAD object 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 | setDefaultConstraints | setBounds | setBudget | setGroups | setGroupRatio | setEquality | setInequality | setTurnover | setOneWayTurnover

Related Examples

- “Setting Default Constraints for Portfolio Weights Using PortfolioMAD Object” on page 6-48
- “Troubleshooting for Setting 'Conditional' BoundType, MinNumAssets, and MaxNumAssets Constraints” on page 4-139
- “Creating the PortfolioMAD Object” on page 6-21
- “Validate the MAD Portfolio Problem” on page 6-76
- “Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object” on page 6-80
- “Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-97
- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-34
- “Working with 'Conditional' BoundType, MinNumAssets, and MaxNumAssets Constraints Using PortfolioMAD Objects” on page 6-67
- “Troubleshooting for Setting 'Conditional' BoundType, MinNumAssets, and MaxNumAssets Constraints” on page 4-139

More About

- “PortfolioMAD Object” on page 6-16
- “Portfolio Optimization Theory” on page 6-2
- “PortfolioMAD Object Workflow” on page 6-15

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 5-10). Budget constraints have properties `LowerBudget` for the lower budget constraint and `UpperBudget` for the upper budget constraint. If you set up a MAD portfolio optimization problem that requires portfolios to be fully invested in your universe of assets, you can set `LowerBudget` to be equal to `UpperBudget`. These budget constraints can be set with default values equal to `1` using `setDefaultConstraints` (see “Setting Default Constraints for Portfolio Weights Using PortfolioMAD Object” on page 6-48).

Setting Budget Constraints Using the PortfolioMAD Function

The properties for the budget constraint can also be set using the `PortfolioMAD` object. 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 6-43. To clear either bound for the budget constraint from your `PortfolioMAD` object, use either the `PortfolioMAD` object 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` | `setDefaultConstraints` | `setBounds` | `setBudget` | `setGroups` | `setGroupRatio` | `setEquality` | `setInequality` | `setTurnover` | `setOneWayTurnover`

Related Examples

- “Setting Default Constraints for Portfolio Weights Using PortfolioMAD Object” on page 6-48
- “Creating the PortfolioMAD Object” on page 6-21
- “Validate the MAD Portfolio Problem” on page 6-76
- “Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object” on page 6-80
- “Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-97
- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-34

More About

- “PortfolioMAD Object” on page 6-16
- “Portfolio Optimization Theory” on page 6-2
- “PortfolioMAD Object Workflow” on page 6-15

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 6-10). 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` object. 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` object, `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` | `setDefaultConstraints` | `setBounds` | `setBudget` | `setGroups` | `setGroupRatio` | `setEquality` | `setInequality` | `setTurnover` | `setOneWayTurnover`

Related Examples

- “Setting Default Constraints for Portfolio Weights Using PortfolioMAD Object” on page 6-48
- “Creating the PortfolioMAD Object” on page 6-21
- “Validate the MAD Portfolio Problem” on page 6-76
- “Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object” on page 6-80
- “Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-97
- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-34

More About

- “PortfolioMAD Object” on page 6-16
- “Portfolio Optimization Theory” on page 6-2
- “PortfolioMAD Object Workflow” on page 6-15

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 6-10). 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` object. 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 nonfinancial companies (assets 4–6). To set group ratio constraints:

```
GA = [ 1 1 1 0 0 0 ];    % financial companies
GB = [ 0 0 0 1 1 1 ];    % nonfinancial companies
p = PortfolioMAD('GroupA', GA, 'GroupB', GB, 'UpperRatio', 0.5);
disp(p.NumAssets)
disp(p.GroupA)
disp(p.GroupB)
disp(p.UpperRatio)
```

```
6

1     1     1     0     0     0

0     0     0     1     1     1

0.5000
```

Group matrices `GA` and `GB` in this example can be logical matrices with `true` and `false` elements that yield the same result:

```
GA = [ true true true false false false ];    % financial companies
GB = [ false false false true true true ];    % nonfinancial companies
p = PortfolioMAD('GroupA', GA, 'GroupB', GB, 'UpperRatio', 0.5);
disp(p.NumAssets)
disp(p.GroupA)
disp(p.GroupB)
disp(p.UpperRatio)
```

```
6

1     1     1     0     0     0

0     0     0     1     1     1

0.5000
```

Setting Group Ratio Constraints Using the setGroupRatio and addGroupRatio 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 nonfinancial companies (assets 4–6). Given a `PortfolioMAD` object `p`, use `setGroupRatio` to set the group constraints:

```
GA = [ true true true false false false ];   % financial companies
GB = [ false false 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.

ThePortfolioMAD object, 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 | setDefaultConstraints | setBounds | setBudget | setGroups | setGroupRatio | setEquality | setInequality | setTurnover | setOneWayTurnover

Related Examples

- “Setting Default Constraints for Portfolio Weights Using PortfolioMAD Object” on page 6-48
- “Creating the PortfolioMAD Object” on page 6-21
- “Validate the MAD Portfolio Problem” on page 6-76
- “Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object” on page 6-80
- “Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-97
- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-34

More About

- “PortfolioMAD Object” on page 6-16
- “Portfolio Optimization Theory” on page 6-2
- “PortfolioMAD Object Workflow” on page 6-15

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 6-8). 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 object. 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` object, `setEquality`, and `addEquality` implement scalar expansion on the `bEquality` property based on the dimension of the matrix in the `AEquality` property.

See Also

`PortfolioMAD` | `setDefaultConstraints` | `setBounds` | `setBudget` | `setGroups` | `setGroupRatio` | `setEquality` | `setInequality` | `setTurnover` | `setOneWayTurnover`

Related Examples

- “Setting Default Constraints for Portfolio Weights Using PortfolioMAD Object” on page 6-48
- “Creating the PortfolioMAD Object” on page 6-21
- “Validate the MAD Portfolio Problem” on page 6-76
- “Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object” on page 6-80
- “Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-97
- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-34

More About

- “PortfolioMAD Object” on page 6-16
- “Portfolio Optimization Theory” on page 6-2
- “PortfolioMAD Object Workflow” on page 6-15

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 6-7). 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` object. 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` object, `setInequality`, and `addInequality` implement scalar expansion on the `bInequality` property based on the dimension of the matrix in the `AInequality` property.

See Also

`PortfolioMAD` | `setDefaultConstraints` | `setBounds` | `setBudget` | `setGroups` | `setGroupRatio` | `setEquality` | `setInequality` | `setTurnover` | `setOneWayTurnover`

Related Examples

- “Setting Default Constraints for Portfolio Weights Using PortfolioMAD Object” on page 6-48
- “Creating the PortfolioMAD Object” on page 6-21
- “Validate the MAD Portfolio Problem” on page 6-76
- “Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object” on page 6-80
- “Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-97
- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-34

More About

- “PortfolioMAD Object” on page 6-16
- “Portfolio Optimization Theory” on page 6-2
- “PortfolioMAD Object Workflow” on page 6-15

Working with 'Conditional' BoundType, MinNumAssets, and MaxNumAssets Constraints Using PortfolioMAD Objects

When any one, or any combination of 'Conditional' BoundType, MinNumAssets, or MaxNumAssets constraints are active, the portfolio problem is formulated by adding NumAssets binary variables, where 0 indicates not invested, and 1 is invested. For example, to explain the 'Conditional' BoundType and MinNumAssets and MaxNumAssets constraints, assume that your portfolio has a universe of 100 assets that you want to invest:

- 'Conditional' BoundType (also known as semicontinuous constraints), set by setBounds, is often used in situations where you do not want to invest small values. A standard example is a portfolio optimization problem where many small allocations are not attractive because of transaction costs. Instead, you prefer fewer instruments in the portfolio with larger allocations. This situation can be handled using'Conditional' BoundType constraints for a PortfolioMAD object.

  For example, the weight you invest in each asset is either 0 or between [0.01, 0.5]. Generally, a semicontinuous variable *x* is a continuous variable between bounds [lb, ub] that also can assume the value 0, where lb > 0, lb ≤ ub. Applying this to portfolio optimization requires that very small or large positions should be avoided, that is values that fall in (0, lb) or are more than ub.
- MinNumAssets and MaxNumAssets (also known as cardinality constraints), set by setMinMaxNumAssets, limit the number of assets in a PortfolioMAD object. For example, if you have 100 assets in your portfolio and you want the number of assets allocated in the portfolio to be from 40 through 60. Using MinNumAssets and MaxNumAssets you can limit the number of assets in the optimized portfolio, which allows you to limit transaction and operational costs or to create an index tracking portfolio.

Setting 'Conditional' BoundType Constraints Using the setBounds Function

Use setBounds with a 'conditional' BoundType to set *xi* = 0 or 0.02 <= *xi* <= 0.5 for all *i*=1,...NumAssets:

```
p = PortfolioMAD;
p = setBounds(p, 0.02, 0.5,'BoundType', 'Conditional', 'NumAssets', 3)
```

```
p =
  Portfolio with properties:

          BuyCost: []
         SellCost: []
     RiskFreeRate: []
        AssetMean: []
       AssetCovar: []
    TrackingError: []
     TrackingPort: []
         Turnover: []
      BuyTurnover: []
     SellTurnover: []
             Name: []
        NumAssets: 3
        AssetList: []
```

```
         InitPort: []
      AInequality: []
      bInequality: []
        AEquality: []
        bEquality: []
       LowerBound: [3×1 double]
       UpperBound: [3×1 double]
      LowerBudget: []
      UpperBudget: []
      GroupMatrix: []
       LowerGroup: []
       UpperGroup: []
           GroupA: []
           GroupB: []
       LowerRatio: []
       UpperRatio: []
        BoundType: [3×1 categorical]
     MinNumAssets: []
     MaxNumAssets: []
```

Setting the Limits on the Number of Assets Invested Using the setMinMaxNumAssets Function

You can also set the `MinNumAssets` and `MaxNumAssets` properties to define a limit on the number of assets invested using `setMinMaxNumAssets`. For example, by setting `MinNumAssets=MaxNumAssets=2`, only two of the three assets are invested in the portfolio.

```
p = PortfolioMAD;
p = setBounds(p, 0.02, 0.5,'BoundType', 'Conditional', 'NumAssets', 3)
p = setMinMaxNumAssets(p, 2, 2)
```

```
 p = 

  PortfolioMAD with properties:

         BuyCost: []
        SellCost: []
    RiskFreeRate: []
        Turnover: []
     BuyTurnover: []
    SellTurnover: []
    NumScenarios: []
            Name: []
       NumAssets: 3
       AssetList: []
        InitPort: []
     AInequality: []
     bInequality: []
       AEquality: []
       bEquality: []
      LowerBound: [3×1 double]
      UpperBound: [3×1 double]
     LowerBudget: []
     UpperBudget: []
     GroupMatrix: []
      LowerGroup: []
      UpperGroup: []
```

```
          GroupA: []
          GroupB: []
      LowerRatio: []
      UpperRatio: []
    MinNumAssets: 2
    MaxNumAssets: 2
       BoundType: [3×1 categorical]
```

See Also

`PortfolioMAD` | `setBounds` | `setMinMaxNumAssets` | `setDefaultConstraints` | `setBounds` | `setBudget` | `setGroups` | `setGroupRatio` | `setEquality` | `setInequality` | `setTurnover` | `setOneWayTurnover`

Related Examples

- “Setting Default Constraints for Portfolio Weights Using PortfolioMAD Object” on page 6-48
- “Working with 'Simple' Bound Constraints Using PortfolioMAD Object” on page 6-52
- “Troubleshooting for Setting 'Conditional' BoundType, MinNumAssets, and MaxNumAssets Constraints” on page 4-139
- “Creating the PortfolioMAD Object” on page 6-21
- “Validate the MAD Portfolio Problem” on page 6-76
- “Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object” on page 6-80
- “Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-97
- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-34

More About

- “PortfolioMAD Object” on page 6-16
- “Portfolio Optimization Theory” on page 6-2
- “PortfolioMAD Object Workflow” on page 6-15

Working with Average Turnover Constraints Using PortfolioMAD Object

The turnover constraint is an optional linear absolute value constraint (see “Average Turnover Constraints” on page 6-11) that enforces an upper bound on the average of purchases and sales. The turnover constraint can be set using the `PortfolioMAD` object 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` object. 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 6-32).

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` object or `setTurnover` with empty inputs for the properties to be cleared.

See Also

`PortfolioMAD` | `setDefaultConstraints` | `setBounds` | `setBudget` | `setGroups` | `setGroupRatio` | `setEquality` | `setInequality` | `setTurnover` | `setOneWayTurnover`

Related Examples

- “Setting Default Constraints for Portfolio Weights Using PortfolioMAD Object” on page 6-48
- “Creating the PortfolioMAD Object” on page 6-21
- “Validate the MAD Portfolio Problem” on page 6-76

- “Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object” on page 6-80
- “Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-97
- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-34

More About

- “PortfolioMAD Object” on page 6-16
- “Portfolio Optimization Theory” on page 6-2
- “PortfolioMAD Object Workflow” on page 6-15

Working with One-Way Turnover Constraints Using PortfolioMAD Object

One-way turnover constraints are optional constraints (see “One-way Turnover Constraints” on page 6-12) that enforce upper bounds on net purchases or net sales. One-way turnover constraints can be set using the PortfolioMAD object 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 object. 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 6-32).

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` object or `setOneWayTurnover` with empty inputs for the properties to be cleared.

See Also

`PortfolioMAD` | `setDefaultConstraints` | `setBounds` | `setBudget` | `setGroups` | `setGroupRatio` | `setEquality` | `setInequality` | `setTurnover` | `setOneWayTurnover`

Related Examples

- “Setting Default Constraints for Portfolio Weights Using PortfolioMAD Object” on page 6-48
- “Creating the PortfolioMAD Object” on page 6-21
- “Validate the MAD Portfolio Problem” on page 6-76
- “Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object” on page 6-80
- “Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-97
- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-34

More About

- "PortfolioMAD Object" on page 6-16
- "Portfolio Optimization Theory" on page 6-2
- "PortfolioMAD Object Workflow" on page 6-15

Validate the MAD Portfolio Problem

In this section...
"Validating a MAD Portfolio Set" on page 6-76
"Validating MAD Portfolios" on page 6-77

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 6-15.

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 PortfolioMAD 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` | `estimateBounds` | `checkFeasibility`

Related Examples

- “Creating the PortfolioMAD Object” on page 6-21
- “Working with MAD Portfolio Constraints Using Defaults” on page 6-48
- “Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object” on page 6-80
- “Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-97
- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-34

More About

- “PortfolioMAD Object” on page 6-16
- “Portfolio Optimization Theory” on page 6-2
- “PortfolioMAD Object Workflow” on page 6-15

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. The “Obtaining Portfolios Along the Entire Efficient Frontier” on page 6-81 example focuses on the former goal and “Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-97 focuses on the latter goal. For information on the workflow when using PortfolioMAD objects, see “PortfolioMAD Object Workflow” on page 6-15.

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 an input argument 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)
```

```
    0.8821    0.7160    0.5498    0.3849    0.2206    0.0530         0         0         0
    0.0429    0.1429    0.2405    0.3392    0.4350    0.5342    0.4706    0.3338    0.1681
    0.0394    0.0426    0.0487    0.0511    0.0559    0.0628    0.0409    0.0029         0
    0.0356    0.0985    0.1609    0.2248    0.2885    0.3499    0.4885    0.6633    0.8319    1.0
```

If you want only four portfolios, you can use `estimateFrontier` with `NumPorts` specified as `4`.

```
pwgt = estimateFrontier(p, 4);
disp(pwgt)
```

```
    0.8821    0.3849         0         0
    0.0429    0.3392    0.4706         0
    0.0394    0.0511    0.0409         0
    0.0356    0.2248    0.4885    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)
```

```
pwgt = 4×10

    0.8821    0.7160    0.5498    0.3849    0.2206    0.0530         0         0         0
    0.0429    0.1429    0.2405    0.3392    0.4350    0.5342    0.4706    0.3338    0.1681
    0.0394    0.0426    0.0487    0.0511    0.0559    0.0628    0.0409    0.0029         0
    0.0356    0.0985    0.1609    0.2248    0.2885    0.3499    0.4885    0.6633    0.8319    1.0
```

```
display(pbuy)
```

```
pbuy = 4×10

    0.5821    0.4160    0.2498    0.0849         0         0         0         0         0
         0         0         0    0.0392    0.1350    0.2342    0.1706    0.0338         0
         0         0         0         0         0         0         0         0         0
         0         0    0.0609    0.1248    0.1885    0.2499    0.3885    0.5633    0.7319    0.9
```

```
display(psell)
```

```
psell = 4×10

         0         0         0         0    0.0794    0.2470    0.3000    0.3000    0.3000    0.3
    0.2571    0.1571    0.0595         0         0         0         0         0    0.1319    0.3
    0.1606    0.1574    0.1513    0.1489    0.1441    0.1372    0.1591    0.1971    0.2000    0.2
    0.0644    0.0015         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

`PortfolioMAD` | `estimateFrontier` | `estimateFrontierLimits` | `estimateFrontierByReturn` | `estimatePortReturn` | `estimateFrontierByRisk` | `estimatePortRisk` | `estimateFrontierByRisk` | `setSolver`

Related Examples

- “Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object” on page 6-80
- “Creating the PortfolioMAD Object” on page 6-21
- “Working with MAD Portfolio Constraints Using Defaults” on page 6-48
- “Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-97
- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-34

More About

- “PortfolioMAD Object” on page 6-16
- “Portfolio Optimization Theory” on page 6-2
- “PortfolioMAD Object Workflow” on page 6-15

Obtaining Endpoints of the Efficient Frontier

Often when using a PortfolioMAD object, 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)

    0.8821         0
    0.0429         0
    0.0394         0
    0.0356    1.0000
```

Note that 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)
```

```
pwgt = 4×2

    0.8860         0
    0.0419         0
    0.0362         0
    0.0359    1.0000
```

```
display(pbuy)
```

```
pbuy = 4×2

    0.5860         0
         0         0
         0         0
         0    0.9000
```

```
display(psell)
```

```
psell = 4×2

         0    0.3000
    0.2581    0.3000
    0.1638    0.2000
    0.0641         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 | estimateFrontierLimits | estimateFrontierByReturn | estimatePortReturn | estimateFrontierByRisk | estimatePortRisk | estimateFrontierByRisk | setSolver

Related Examples

- “Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object” on page 6-80
- “Creating the PortfolioMAD Object” on page 6-21
- “Working with MAD Portfolio Constraints Using Defaults” on page 6-48
- “Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-97
- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-34

More About

- “PortfolioMAD Object” on page 6-16
- “Portfolio Optimization Theory” on page 6-2
- “PortfolioMAD Object Workflow” on page 6-15

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 6-83 and “Obtaining MAD Portfolio Risks and Returns” on page 6-97).

```
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` | `estimateFrontierLimits` | `estimateFrontierByReturn` | `estimatePortReturn` | `estimateFrontierByRisk` | `estimatePortRisk` | `estimateFrontierByRisk` | `setSolver`

Related Examples

- “Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object” on page 6-80
- “Creating the PortfolioMAD Object” on page 6-21
- “Working with MAD Portfolio Constraints Using Defaults” on page 6-48
- “Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-97
- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-34

More About

- “PortfolioMAD Object” on page 6-16

- “Portfolio Optimization Theory” on page 6-2
- “PortfolioMAD Object Workflow” on page 6-15

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 6-83 and “Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-97).

```
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` | `estimateFrontierLimits` | `estimateFrontierByReturn` | `estimatePortReturn` | `estimateFrontierByRisk` | `estimatePortRisk` | `estimateFrontierByRisk` | `setSolver`

Related Examples

- "Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object" on page 6-80
- "Creating the PortfolioMAD Object" on page 6-21
- "Working with MAD Portfolio Constraints Using Defaults" on page 6-48
- "Estimate Efficient Frontiers for PortfolioMAD Object" on page 6-97
- "Asset Returns and Scenarios Using PortfolioMAD Object" on page 6-34

More About

- “PortfolioMAD Object” on page 6-16
- “Portfolio Optimization Theory” on page 6-2
- “PortfolioMAD Object Workflow” on page 6-15

Choosing and Controlling the Solver for PortfolioMAD Optimizations

When solving portfolio optimizations for a PortfolioMAD object, you are solving nonlinear optimization problems with either nonlinear objective or nonlinear constraints. You can use 'TrustRegionCP' (default) or 'ExtendedCP' solvers that implement Kelley’s cutting plane method (see Kelley [45] at “Portfolio Optimization” on page A-5). Alternatively, you can use fmincon and all variations of fmincon from Optimization Toolbox are supported. When using fmincon as the solverType, 'sqp' is the default algorithm for fmincon.

Using 'TrustRegionCP' and 'ExtendedCP' SolverTypes

The 'TrustRegionCP' and 'ExtendedCP' solvers have options to control the number iterations and stopping tolerances. Moreover, these solvers use linprog as the primary 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 'TrustRegionCP':

```
p = PortfolioMAD;
p = setSolver(p, 'TrustRegionCP', 'MaxIterations', 2000);
display(p.solverType)
display(p.solverOptions)
```

```
trustregioncp
                MaxIterations: 2000
         AbsoluteGapTolerance: 1.0000e-07
         RelativeGapTolerance: 1.0000e-05
       NonlinearScalingFactor: 1000
       ObjectiveScalingFactor: 1000
          MainSolverOptions: [1×1 optim.options.Linprog]
                      Display: 'off'
                CutGeneration: 'basic'
     MaxIterationsInactiveCut: 30
           ActiveCutTolerance: 1.0000e-07
                  ShrinkRatio: 0.7500
    TrustRegionStartIteration: 2
                   DeltaLimit: 1
```

To change the primary solver algorithm to 'interior-point', with no display, use setSolver to modify 'MainSolverOptions':

```
p = PortfolioMAD;
options = optimoptions('linprog','Algorithm','interior-point','Display','off');
p = setSolver(p,'TrustRegionCP','MainSolverOptions',options);
display(p.solverType)
display(p.solverOptions)
display(p.solverOptions.MainSolverOptions.Algorithm)
display(p.solverOptions.MainSolverOptions.Display)
```

```
trustregioncp
                MaxIterations: 1000
         AbsoluteGapTolerance: 1.0000e-07
         RelativeGapTolerance: 1.0000e-05
       NonlinearScalingFactor: 1000
       ObjectiveScalingFactor: 1000
          MainSolverOptions: [1×1 optim.options.Linprog]
```

```
                      Display: 'off'
                CutGeneration: 'basic'
     MaxIterationsInactiveCut: 30
           ActiveCutTolerance: 1.0000e-07
                  ShrinkRatio: 0.7500
    TrustRegionStartIteration: 2
                   DeltaLimit: 1

interior-point
off
```

Using 'fmincon' SolverType

Unlike Optimization Toolbox which uses the `'interior-point'` algorithm as the default algorithm for `fmincon`, the portfolio optimization for a `PortfolioMAD` object uses the `'sqp'` algorithm as the default. For details about `fmincon` and constrained nonlinear optimization algorithms and options, see “Constrained Nonlinear Optimization Algorithms”.

To modify `fmincon` options for MAD portfolio optimizations, use `setSolver` to set the hidden properties `solverType` and `solverOptions` to specify and control the solver. Since these solver properties are hidden, you cannot set them using the `PortfolioMAD` object. The default for the `fmincon` solver is the `'sqb'` algorithm and no displayed output, so you do not need to use `setSolver` to specify the `'sqp'` algorithm for `fmincon`.

```
p = PortfolioMAD;
p = setSolver(p, 'fmincon');
display(p.solverOptions.Algorithm)
display(p.solverOptions.Display)
```

```
sqp
off
```

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 `'active-set'` algorithm and with displayed output, use `setSolver` with:

```
p = PortfolioMAD;
p = setSolver(p, 'fmincon', 'Algorithm', 'active-set', 'Display', 'final');
display(p.solverOptions)
```

```
fmincon options:

   Options used by current Algorithm ('active-set'):
   (Other available algorithms: 'interior-point', 'sqp', 'sqp-legacy', 'trust-region-reflective'

   Set properties:
                    Algorithm: 'active-set'
                      Display: 'final'

   Default properties:
               CheckGradients: 0
          ConstraintTolerance: 1.0000e-06
     FiniteDifferenceStepSize: 'sqrt(eps)'
         FiniteDifferenceType: 'forward'
            FunctionTolerance: 1.0000e-06
       MaxFunctionEvaluations: '100*numberOfVariables'
                MaxIterations: 400
          OptimalityTolerance: 1.0000e-06
                    OutputFcn: []
```

```
                      PlotFcn: []
    SpecifyConstraintGradient: 0
     SpecifyObjectiveGradient: 0
                StepTolerance: 1.0000e-06
                     TypicalX: 'ones(numberOfVariables,1)'
                  UseParallel: 0
```

Alternatively, the `setSolver` function accepts an `optimoptions` object as the second argument. For example, you can change the algorithm to `'active-set'` with no displayed output as follows:

```
p = PortfolioMAD;
options = optimoptions('fmincon', 'Algorithm', 'active-set', 'Display', 'off');
p = setSolver(p, 'fmincon', options);
display(p.solverOptions.Algorithm)
display(p.solverOptions.Display)
```

```
active-set
off
```

Using the Mixed Integer Nonlinear Programming (MINLP) Solver

The mixed integer nonlinear programming (MINLP) solver, configured using `setSolverMINLP`, enables you to specify associated solver options for portfolio optimization for a `PortfolioMAD` object. The MINLP solver is used when any one, or any combination of `'Conditional'` `BoundType`, `MinNumAssets`, or `MaxNumAssets` constraints are active, the portfolio problem is formulated by adding `NumAssets` binary variables, where `0` indicates not invested, and `1` is invested. For more information on using `'Conditional'` `BoundType`, see `setBounds`. For more information on specifying `MinNumAssets` and `MaxNumAssets`, see `setMinMaxNumAssets`.

When using the `estimate` functions with a `PortfolioMAD` object where `'Conditional'` `BoundType`, `MinNumAssets`, or `MaxNumAssets` constraints are active, the mixed integer nonlinear programming (MINLP) solver is automatically used.

Solver Guidelines for PortfolioMAD Objects

The following table provides guidelines for using `setSolver` and `setSolverMINLP`.

PortfolioMAD Problem	PortfolioMAD Function	Type of Optimization Problem	Main Solver	Helper Solver
PortfolioMAD without active `'Conditional'` `BoundType`, `MinNumAssets`, and `MaxNumAssets`	`estimateFrontierByRisk`	Optimizing a portfolio for a certain risk level introduces a nonlinear constraint. Therefore, this problem has a linear objective with linear and nonlinear constraints.	`'TrustRegionCP'`, `'ExtendedCP'`, or `'fmincon'` using `setSolver`	`'linprog'` using `setSolver`

PortfolioMAD Problem	PortfolioMAD Function	Type of Optimization Problem	Main Solver	Helper Solver
PortfolioMAD without active 'Conditional' BoundType, MinNumAssets, and MaxNumAssets	estimateFrontierByReturn	Nonlinear objective with linear constraints	'TrustRegionCP', 'ExtendedCP', or 'fmincon' using setSolver	'linprog' using setSolver
PortfolioMAD without active 'Conditional' BoundType, MinNumAssets, and MaxNumAssets	estimateFrontierLimits	Nonlinear or linear objective with linear constraints	For 'min': nonlinear objective, 'TrustRegionCP', 'ExtendedCP', or 'fmincon' using setSolver For 'max': linear objective, 'linprog' using setSolver	Not applicable
PortfolioMAD with active 'Conditional' BoundType, MinNumAssets, and MaxNumAssets	estimateFrontierByRisk	The problem is formulated by introducing NumAssets binary variables to indicate whether the corresponding asset is invested or not. Therefore, it requires a mixed integer nonlinear programming solver. Three types of MINLP solvers are offered, see setSolverMINLP.	Mixed integer nonlinear programming solver (MINLP) using setSolverMINLP	'fmincon' is used when the estimate functions reduce the problem into NLP. This solver is configured through setSolver.

PortfolioMAD Problem	PortfolioMAD Function	Type of Optimization Problem	Main Solver	Helper Solver
PortfolioMAD with active `'Conditional'` `BoundType`, `MinNumAssets`, and `MaxNumAssets`	`estimateFrontierByReturn`	The problem is formulated by introducing `NumAssets` binary variables to indicate whether the corresponding asset is invested or not. Therefore, it requires a mixed integer nonlinear programming solver. Three types of MINLP solvers are offered, see `setSolverMINLP`.	Mixed integer nonlinear programming solver (MINLP) using `setSolverMINLP`	`'fmincon'` is used when the `estimate` functions reduce the problem into NLP. This solver is configured through `setSolver`
PortfolioMAD with active `'Conditional'` `BoundType`, `MinNumAssets`, and `MaxNumAssets`	`estimateFrontierLimits`	The problem is formulated by introducing `NumAssets` binary variables to indicate whether the corresponding asset is invested or not. Therefore, it requires a mixed integer nonlinear programming solver. Three types of MINLP solvers are offered, see `setSolverMINLP`.	Mixed integer nonlinear programming solver (MINLP) using `setSolverMINLP`	`'fmincon'` is used when the `estimate` functions reduce the problem into NLP. This solver is configured through `setSolver`

See Also

`PortfolioMAD` | `estimateFrontier` | `estimateFrontierLimits` | `estimateFrontierByReturn` | `estimatePortReturn` | `estimateFrontierByRisk` | `estimatePortRisk` | `estimateFrontierByRisk` | `setSolver` | `setSolverMINLP`

Related Examples

- “Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object” on page 6-80
- “Creating the PortfolioMAD Object” on page 6-21
- “Working with MAD Portfolio Constraints Using Defaults” on page 6-48
- “Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-97
- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-34

- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-183

More About

- “PortfolioMAD Object” on page 6-16
- “Portfolio Optimization Theory” on page 6-2
- “PortfolioMAD Object Workflow” on page 6-15
- “Working with 'Conditional' BoundType, MinNumAssets, and MaxNumAssets Constraints Using Portfolio Objects” on page 4-78

Estimate Efficient Frontiers for PortfolioMAD Object

In this section...
"Obtaining MAD Portfolio Risks and Returns" on page 6-97
"Obtaining the PortfolioMAD Standard Deviation" on page 6-98

Whereas "Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object" on page 6-80 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 6-15.

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 | plotFrontier | estimatePortStd

Related Examples

- “Plotting the Efficient Frontier for a PortfolioMAD Object” on page 6-100
- “Creating the PortfolioMAD Object” on page 6-21
- “Working with MAD Portfolio Constraints Using Defaults” on page 6-48
- “Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object” on page 6-80
- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-34
- “Postprocessing Results to Set Up Tradable Portfolios” on page 6-105

More About

- “PortfolioMAD Object” on page 6-16
- “Portfolio Optimization Theory” on page 6-2
- “PortfolioMAD Object Workflow” on page 6-15

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 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)
```

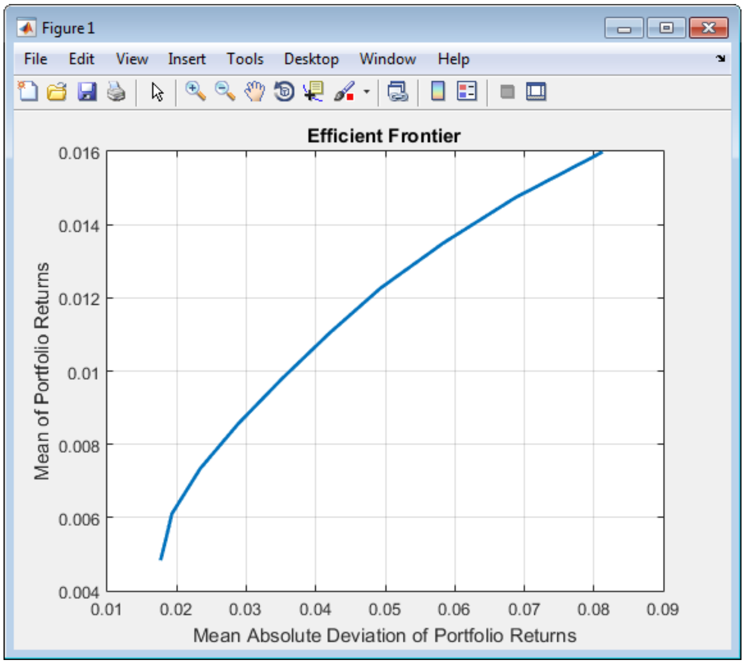


Figure 1
File Edit View Insert Tools Desktop Window Help
Efficient Frontier
Mean of Portfolio Returns
0.016
0.014
0.012
0.01
0.008
0.006
0.004
0.01 0.02 0.03 0.04 0.05 0.06 0.07 0.08 0.09
Mean Absolute Deviation of Portfolio Returns

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
```

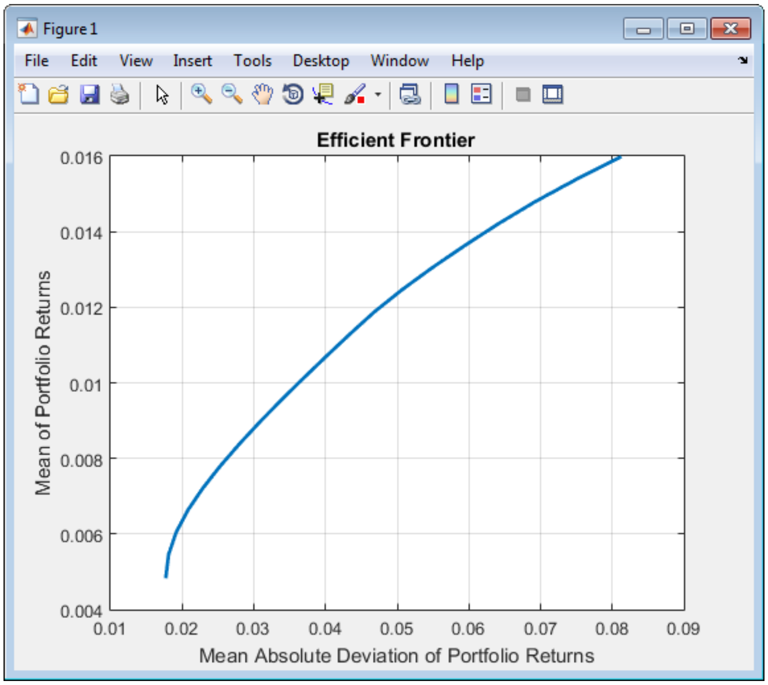
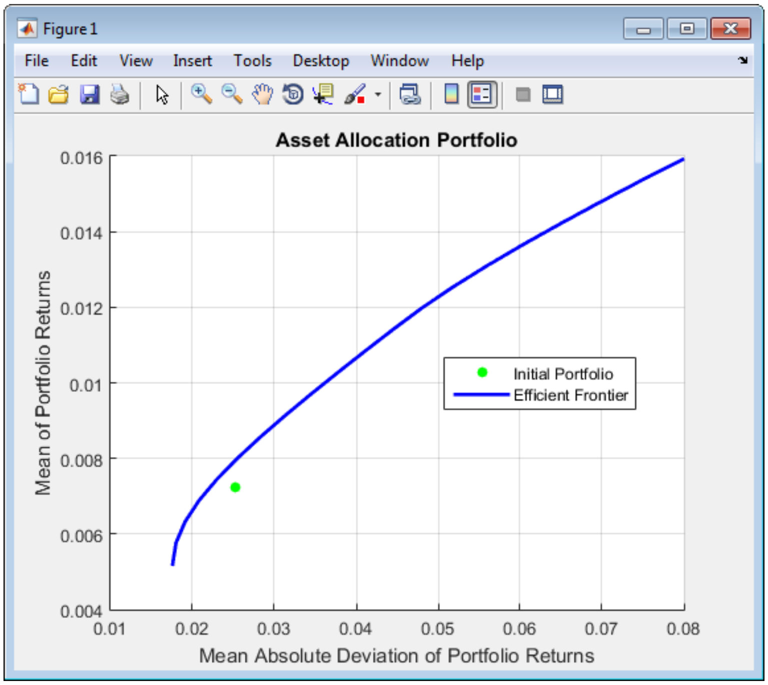


Figure 1
File Edit View Insert Tools Desktop Window Help
Efficient Frontier
0.016
0.014
0.012
0.01
0.008
0.006
0.004
Mean of Portfolio Returns
0.01 0.02 0.03 0.04 0.05 0.06 0.07 0.08 0.09
Mean Absolute Deviation of Portfolio Returns

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 6-97), 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)
```



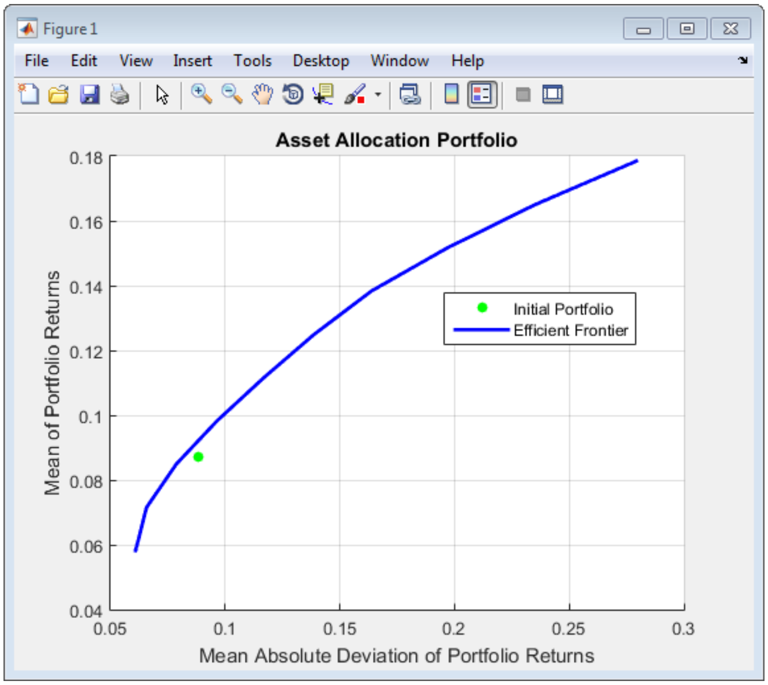
Plotting Existing Efficient Portfolio Risks and Returns

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` | `plotFrontier` | `estimatePortStd`

Related Examples

- “Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-97
- “Creating the PortfolioMAD Object” on page 6-21
- “Working with MAD Portfolio Constraints Using Defaults” on page 6-48
- “Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object” on page 6-80
- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-34
- “Postprocessing Results to Set Up Tradable Portfolios” on page 6-105

More About

- “PortfolioMAD Object” on page 6-16
- “Portfolio Optimization Theory” on page 6-2
- “PortfolioMAD Object Workflow” on page 6-15

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 6-15.

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` | `estimateScenarioMoments` | `checkFeasibility`

Related Examples

- “Creating the PortfolioMAD Object” on page 6-21
- “Working with MAD Portfolio Constraints Using Defaults” on page 6-48
- “Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object” on page 6-80
- “Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-97
- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-34

More About

- “PortfolioMAD Object” on page 6-16
- “Portfolio Optimization Theory” on page 6-2
- “PortfolioMAD Object Workflow” on page 6-15

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

`PortfolioMAD` | `Portfolio`

Related Examples

- “Creating the Portfolio Object” on page 4-24
- “Creating the PortfolioMAD Object” on page 6-21
- “Working with MAD Portfolio Constraints Using Defaults” on page 6-48
- “Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object” on page 6-80
- “Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-97
- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-34

More About

- “PortfolioMAD Object” on page 6-16
- “Portfolio Optimization Theory” on page 6-2
- “PortfolioMAD Object Workflow” on page 6-15

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 object if you want to modify it, otherwise it creates a new object. See "Creating the PortfolioMAD Object" on page 6-21 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 6-19.

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 6-76. 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` | `estimateScenarioMoments` | `checkFeasibility`

Related Examples

- “Troubleshooting for Setting 'Conditional' BoundType, MinNumAssets, and MaxNumAssets Constraints” on page 4-139
- “Postprocessing Results to Set Up Tradable Portfolios” on page 6-105
- “Creating the PortfolioMAD Object” on page 6-21
- “Working with MAD Portfolio Constraints Using Defaults” on page 6-48
- “Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object” on page 6-80
- “Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-97
- “Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-34

More About

- “PortfolioMAD Object” on page 6-16
- “Portfolio Optimization Theory” on page 6-2
- “PortfolioMAD Object Workflow” on page 6-15

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. Drawdown is the peak to trough decline during a specific record period of an investment or fund. To calculate maximum or expected maximum drawdowns, use `maxdrawdown` and `emaxdrawdown`.

See Also

`sharpe` | `inforatio` | `portalpha` | `lpm` | `elpm` | `maxdrawdown` | `emaxdrawdown` | `ret2tick` | `tick2ret`

Related Examples

- “Using the Sharpe Ratio” on page 7-5
- “Using the Information Ratio” on page 7-7
- “Using Tracking Error” on page 7-9
- “Using Risk-Adjusted Return” on page 7-10
- “Using Sample and Expected Lower Partial Moments” on page 7-12
- “Using Maximum and Expected Maximum Drawdown” on page 7-14

Performance Metrics Illustration

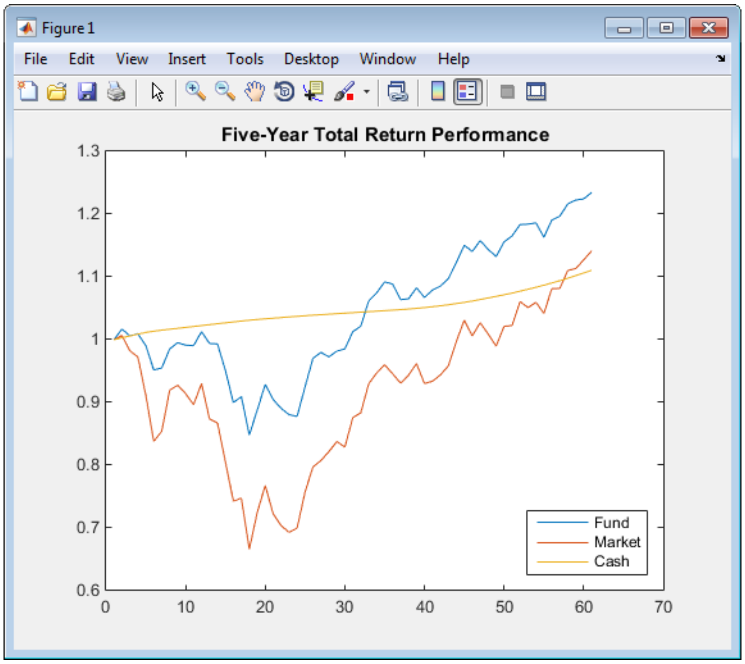
To illustrate the functions for investment performance metrics, you work with three financial time series objects using performance data for:

- 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

`sharpe` | `inforatio` | `portalpha` | `lpm` | `elpm` | `maxdrawdown` | `emaxdrawdown` | `ret2tick` | `tick2ret`

Related Examples

- “Using the Sharpe Ratio” on page 7-5
- “Using the Information Ratio” on page 7-7
- “Using Tracking Error” on page 7-9
- “Using Risk-Adjusted Return” on page 7-10
- “Using Sample and Expected Lower Partial Moments” on page 7-12
- “Using Maximum and Expected Maximum Drawdown” on page 7-14

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 4-107.

See Also

`sharpe` | `inforatio` | `portalpha` | `lpm` | `elpm` | `maxdrawdown` | `emaxdrawdown` | `ret2tick` | `tick2ret` | `Portfolio`

Related Examples

- “Performance Metrics Overview” on page 7-2
- “Using the Information Ratio” on page 7-7
- “Using Tracking Error” on page 7-9
- “Using Risk-Adjusted Return” on page 7-10
- “Using Sample and Expected Lower Partial Moments” on page 7-12
- “Using Maximum and Expected Maximum Drawdown” on page 7-14

Using the Information Ratio

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

`sharpe` | `inforatio` | `portalpha` | `lpm` | `elpm` | `maxdrawdown` | `emaxdrawdown` | `ret2tick` | `tick2ret`

Related Examples

- "Performance Metrics Overview" on page 7-2
- "Using the Sharpe Ratio" on page 7-5

- “Using Tracking Error” on page 7-9
- “Using Risk-Adjusted Return” on page 7-10
- “Using Sample and Expected Lower Partial Moments” on page 7-12
- “Using Maximum and Expected Maximum Drawdown” on page 7-14

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, also know as active risk, measures the volatility of active returns. 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

`sharpe` | `inforatio` | `portalpha` | `lpm` | `elpm` | `maxdrawdown` | `emaxdrawdown` | `ret2tick` | `tick2ret`

Related Examples

- “Performance Metrics Overview” on page 7-2
- “Using the Sharpe Ratio” on page 7-5
- “Using the Information Ratio” on page 7-7
- “Using Risk-Adjusted Return” on page 7-10
- “Using Sample and Expected Lower Partial Moments” on page 7-12
- “Using Maximum and Expected Maximum Drawdown” on page 7-14

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

`sharpe` | `inforatio` | `portalpha` | `lpm` | `elpm` | `maxdrawdown` | `emaxdrawdown` | `ret2tick` | `tick2ret`

Related Examples

- “Performance Metrics Overview” on page 7-2
- “Using the Sharpe Ratio” on page 7-5
- “Using the Information Ratio” on page 7-7
- “Using Tracking Error” on page 7-9
- “Using Sample and Expected Lower Partial Moments” on page 7-12
- “Using Maximum and Expected Maximum Drawdown” on page 7-14

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

`sharpe` | `inforatio` | `portalpha` | `lpm` | `elpm` | `maxdrawdown` | `emaxdrawdown` | `ret2tick` | `tick2ret`

Related Examples

- “Performance Metrics Overview” on page 7-2
- “Using the Sharpe Ratio” on page 7-5
- “Using the Information Ratio” on page 7-7
- “Using Tracking Error” on page 7-9
- “Using Risk-Adjusted Return” on page 7-10
- “Using Maximum and Expected Maximum Drawdown” on page 7-14

Using Maximum and Expected Maximum Drawdown

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 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 is 16.58% for the fund series and 33.81% for the market. There is 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 NaNs because it never has a drawdown.

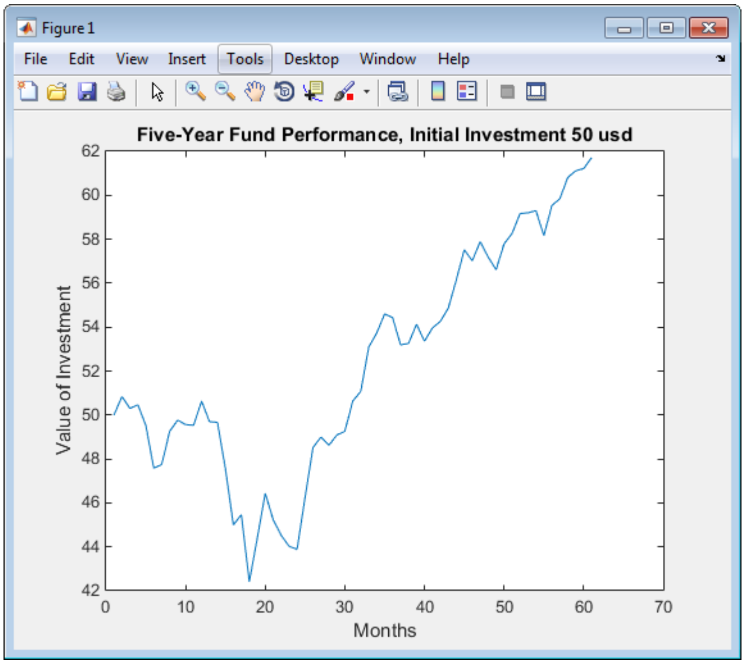
The 16.58% value loss from month 2 to month 18 for the fund series is verified using the reported indices:

```
Start = MaxDDIndex(1,:);
End = MaxDDIndex(2,:);
(TestData(Start(1),1) - TestData(End(1),1))/TestData(Start(1),1)
```

```
ans =

    0.1658
```

Although the maximum drawdown is measured in terms of returns, `maxdrawdown` can measure the drawdown in terms of absolute drop in value, or in terms of log-returns. To contrast these alternatives more clearly, you can 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, 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, 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 is $50.84 in month 2, but by month 18 the value is 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.

Finally, you can 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, Sigma, T)
```

which gives the following results:

```
MaxDD =

    0.1813
```

```
EMaxDD =

    0.1545
```

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

`sharpe` | `inforatio` | `portalpha` | `lpm` | `elpm` | `maxdrawdown` | `emaxdrawdown` | `ret2tick` | `tick2ret`

Related Examples

- “Performance Metrics Overview” on page 7-2
- “Using the Sharpe Ratio” on page 7-5
- “Using the Information Ratio” on page 7-7
- “Using Tracking Error” on page 7-9
- “Using Risk-Adjusted Return” on page 7-10
- “Using Sample and Expected Lower Partial Moments” on page 7-12

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 one-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
```

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 five-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 one-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 two-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'}`.

Working with a Transition Matrix Containing NR Rating

This example demonstrates how `'NR'` (not rated) ratings are handled by `transprob`, and how to get transition matrix that use the `'NR'` rating information for the estimation, but that do not show the `'NR'` rating in the final transition probabilities.

The dataset `data` from `Data_TransProb.mat` contains sample credit ratings using the default rating scale `{'AAA', 'AA','A', 'BBB', 'BB', 'B', 'CCC', 'D'}`.

```
load Data_TransProb
head(data,12)
```

```
ans =

  12×3 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'  
    '00014413'    '25-Nov-1999'    'BB' 
    '00012126'    '17-Feb-1985'    'CCC'
```

Replace a transition to `'B'` with a transition to `'NR'` for the first company. Note that there is a subsequent transition from `'NR'` to `'CCC'`.

```
dataNR = data;
dataNR.Rating{2} = 'NR';
dataNR.Rating{7} = 'NR';

head(dataNR,12)

ans =

  12×3 table

        ID              Date          Rating
    __________    _____________    ______

    '00010283'    '10-Nov-1984'    'CCC'
    '00010283'    '12-May-1986'    'NR' 
    '00010283'    '29-Jun-1988'    'CCC'
    '00010283'    '12-Dec-1991'    'D'  
    '00013326'    '09-Feb-1985'    'A'  
    '00013326'    '24-Feb-1994'    'AA' 
    '00013326'    '10-Nov-2000'    'NR' 
    '00014413'    '23-Dec-1982'    'B'  
    '00014413'    '20-Apr-1988'    'BB' 
    '00014413'    '16-Jan-1998'    'B'  
    '00014413'    '25-Nov-1999'    'BB' 
    '00012126'    '17-Feb-1985'    'CCC'
```

`'NR'` is treated as another rating. The transition matrix shows the estimated probability of transitioning into and out of `'NR'`. In this example, the `transprob` function uses the`'cohort'` algorithm, and the `'NR'` rating is treated as another rating. The same behavior exists when using the `transprob` function with the `'duration'` algorithm.

```
RatingsLabelsNR = {'AAA','AA','A','BBB','BB','B','CCC','D','NR'};
[MatrixNRCohort,TotalsNRCohort] = transprob(dataNR,...
   'Labels',RatingsLabelsNR,...
   'Algorithm','cohort');

fprintf('Transition probability, cohort, including NR:\n')
disp(array2table(MatrixNRCohort,'VariableNames',RatingsLabelsNR,...
   'RowNames',RatingsLabelsNR))

fprintf('Total transitions out of given rating, including 6 out of NR (5 NR->NR, 1 NR->CCC):\n')
disp(array2table(TotalsNRCohort.totalsVec,'VariableNames',RatingsLabelsNR))

Transition probability, cohort, including NR:
             AAA        AA          A          BBB         BB          B          CCC
           ________    _______    ________    _______    ________    ________    ________    ____
```

```
    AAA      93.135     5.9335     0.74557    0.15533    0.031066           0           0
    AA       1.7359      92.92      4.5446    0.58514     0.15604           0           0     0.03
    A        0.12683     2.9716     91.991     4.3124      0.4711    0.054358           0     0.07
    BBB     0.021048    0.37887     5.0726     89.771      4.0413     0.46306    0.042096      0.2
    BB      0.022099     0.1105    0.68508      6.232      88.376      3.6464     0.28729      0.6
    B              0          0   0.076161    0.72353       7.997      86.215      2.7037       2
    CCC            0          0          0    0.30936      1.8561      4.4857      80.897       12
    D              0          0          0          0           0           0           0
    NR             0          0          0          0           0           0      16.667

Total transitions out of given rating, including 6 out of NR (5 NR->NR, 1 NR->CCC):
    AAA      AA       A      BBB      BB       B      CCC       D      NR 
    _____   _____   _____   _____   _____   _____   _____   _____   __

    3219    5127    5519    4751    4525    2626    1293    4050    6 
```

To remove transitions to `'NR'` from the transition matrix, you need to use the `'excludeLabels'` optional name-value input argument to `transprob`.

The `'labels'` input to `transprob` may or may not include the label that needs to be excluded. In the following example, the NR rating is removed from the labels for display purposes, but passing `RatingsLabelsNR` to `transprob` would also work.

```
RatingsLabels = {'AAA','AA','A','BBB','BB','B','CCC','D'};
```

```
[MatrixCohort,TotalsCohort] = transprob(dataNR,'Labels',RatingsLabels,'ExcludeLabels','NR','Algo
```

```
fprintf('Transition probability, cohort, after postprocessing to remove NR:\n')
```

```
Transition probability, cohort, after postprocessing to remove NR:
```

```
disp(array2table(MatrixCohort,'VariableNames',RatingsLabels,...
   'RowNames',RatingsLabels))
```

```
Transition probability, cohort, after postprocessing to remove NR:
             AAA         AA          A          BBB         BB          B          CCC          D
           ________    _______    ________    _______    ________    ________    ________     ___

    AAA      93.135     5.9335     0.74557    0.15533    0.031066           0           0
    AA       1.7362     92.938      4.5455    0.58525     0.15607           0           0     0.03
    A        0.12683     2.9716     91.991     4.3124      0.4711    0.054358           0     0.07
    BBB     0.021048    0.37887     5.0726     89.771      4.0413     0.46306    0.042096      0.2
    BB      0.022099     0.1105    0.68508      6.232      88.376      3.6464     0.28729      0.6
    B              0          0   0.076161    0.72353       7.997      86.215      2.7037       2
    CCC            0          0          0     0.3096      1.8576      4.4892       80.96       12
    D              0          0          0          0           0           0           0

Total transitions out of given rating, AA and CCC have one less than before:
    AAA      AA       A      BBB      BB       B      CCC       D  
    _____   _____   _____   _____   _____   _____   _____   _____

    3219    5126    5519    4751    4525    2626    1292    4050 
```

```
fprintf('Total transitions out of given rating, AA and CCC have one less than before:\n')
```

```
Total transitions out of given rating, AA and CCC have one less than before
```

```
disp(array2table(TotalsCohort.totalsVec,'VariableNames',RatingsLabels))
```

```
    AAA     AA      A      BBB     BB      B      CCC      D
    ____    ____    ____    ____    ____    ____    ____    ____

    3219    5126    5519    4751    4525    2626    1292    4050
```

All transitions involving `'NR'` are removed from the sample, but all other transitions are still used to estimate the transition probabilities. In this example, the transition from `'NR'` to `'CCC'` has been removed, as well as the transition from `'AA'` to `'NR'` (and five more transitions from `'NR'` to `'NR'`). That means the first company is still contributing transitions from`'CCC'` to `'CCC'` for the estimation, only the periods overlapping with the time this company spent in `'NR'` have been removed from the sample, and similarly for the other company.

This procedure is different from removing the `'NR'` rows from the data itself.

For example, if you remove the `'NR'` rows in this example, the first company seems to stay in its initial rating of `'CCC'` all the way from the initial date in 1984 to the default event in 1991. With the previous approach, the estimation knows that the company transitioned out of `'CCC'` at some point, it knows it was not staying at `'CCC'` all the time.

If the `'NR'` row is removed for the second company, this company seems to have stayed in the sample as an `'AA'` company until the end of the sample. With the previous approach, the estimation knows that this company stopped being an `'AA'` earlier.

```
dataNR2 = dataNR;
dataNR2([2 7],:) = [];

head(dataNR2,12)
```

```
ans =

  12×3 table

        ID              Date           Rating
    __________    _____________    ______

    '00010283'    '10-Nov-1984'    'CCC'
    '00010283'    '29-Jun-1988'    'CCC'
    '00010283'    '12-Dec-1991'    'D'
    '00013326'    '09-Feb-1985'    'A'
    '00013326'    '24-Feb-1994'    'AA'
    '00014413'    '23-Dec-1982'    'B'
    '00014413'    '20-Apr-1988'    'BB'
    '00014413'    '16-Jan-1998'    'B'
    '00014413'    '25-Nov-1999'    'BB'
    '00012126'    '17-Feb-1985'    'CCC'
    '00012126'    '08-Mar-1989'    'D'
    '00011692'    '11-May-1984'    'BB'
```

If the `'NR'` rows are removed, the transition matrices will be different. The probability of staying at `'CCC'` goes slightly up, and so does the probability of staying at `'AA'`.

The transition matrices will be different. The probability of staying at `'CCC'` goes slightly up, and so does the probability of staying at `'AA'`.

```
[MatrixCohort2,TotalsCohort2] = transprob(dataNR2,...
   'Labels',RatingsLabels,...
   'Algorithm','cohort');
```

```
fprintf('Transition probability, cohort, if NR rows are removed from data:\n')
disp(array2table(MatrixCohort2,'VariableNames',RatingsLabels,...
   'RowNames',RatingsLabels))

fprintf('Total transitions out of given rating, many more out of CCC and AA:\n')
disp(array2table(TotalsCohort2.totalsVec,'VariableNames',RatingsLabels))
```

```
Transition probability, cohort, if NR rows are removed from data:
```

```
disp(array2table(MatrixCohort2,'VariableNames',RatingsLabels,...
   'RowNames',RatingsLabels))
```

```
Transition probability, cohort, if NR rows are removed from data:
             AAA         AA          A          BBB         BB          B          CCC          D
           ________    _______    ________    _______    ________    ________    ________    ___

    AAA      93.135     5.9335     0.74557    0.15533    0.031066           0           0
    AA       1.7346     92.945       4.541    0.58468     0.15592           0           0    0.03
    A        0.12683     2.9716      91.991     4.3124      0.4711    0.054358           0    0.07
    BBB    0.021048    0.37887      5.0726     89.771      4.0413     0.46306    0.042096     0.2
    BB     0.022099     0.1105     0.68508      6.232      88.376      3.6464     0.28729     0.6
    B              0          0    0.076161    0.72353       7.997      86.215      2.7037      2
    CCC            0          0           0    0.30888      1.8533      4.4788      81.004      12
    D              0          0           0          0           0           0           0
```

```
fprintf('Total transitions out of given rating, many more out of CCC and AA:\n')
```

```
Total transitions out of given rating, many more out of CCC and AA:
```

```
disp(array2table(TotalsCohort2.totalsVec,'VariableNames',RatingsLabels))
```

```
    AAA      AA      A      BBB      BB      B      CCC      D 
    _____   _____   _____   _____   _____   _____   _____   _____

    3219    5131    5519    4751    4525    2626    1295    4050
```

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 one-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, one-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 8-9), but idTotals has the information at an ID level. Because most companies only migrate between 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-'`, and so on), whole ratings (`'AA'`, `'BB'`, and so on), 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 8-12).

```
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

`transprob` | `transprobprep` | `transprobbytotals` | `bootstrp` | `transprobgrouptotals` | `transprobtothresholds` | `transprobfromthresholds`

Related Examples

- “Credit Quality Thresholds” on page 8-43
- “Credit Rating by Bagging Decision Trees”
- “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 on page 8-42]. 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 on page 8-42], and a corporate spread, from [4 on page 8-41] (variables `CPF` and `SPR`). A variable indicating recession years (`Recession`), consistent with recession dates from [7 on page 8-42], 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.

Use functionality in Financial Toolbox™, specifically, the 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.

Use `transprobgrouptotals` to group the 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. 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'})
```

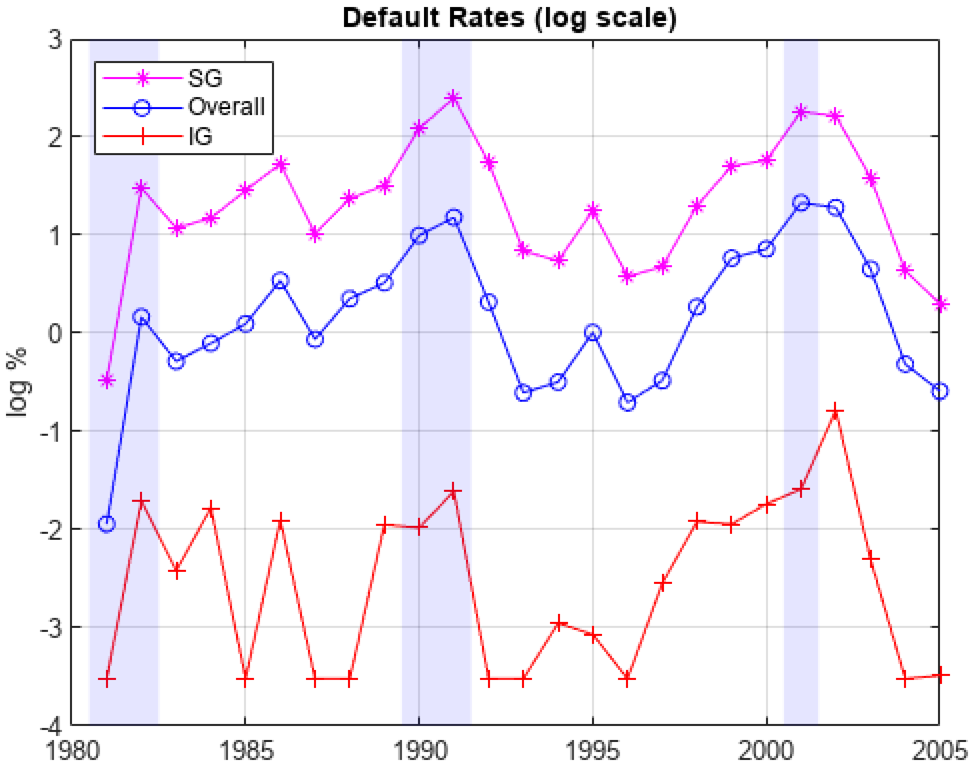
```
           IG     SG      D     NR
    IG  92.98   1.65   0.03   5.33
    SG   3.07  82.75   1.35  12.83
```

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

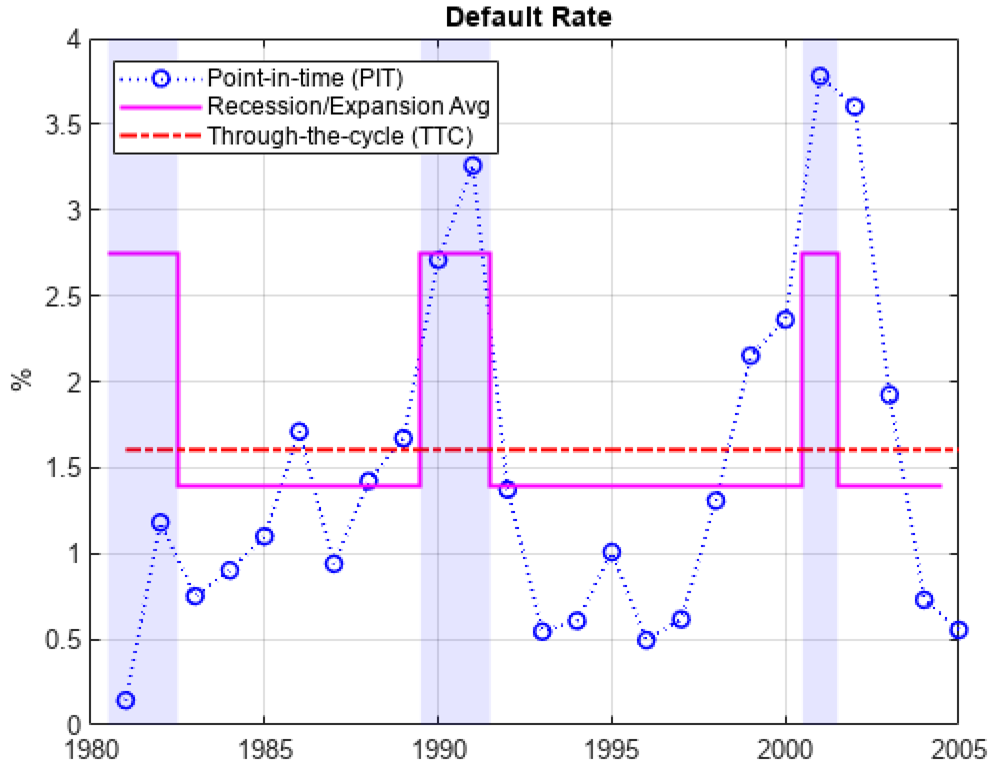
The default rates obtained are examples of point-in-time (PIT) rates, only the most recent information is used to estimate them. On the other extreme, we can use all the migrations observed in the 25 years spanned by the dataset to estimate long-term, or through-the-cycle (TTC) default rates. Other rates 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 on page 8-42]) 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, for example, [1 on page 8-41] and [5 on page 8-41]. 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 on page 8-42], 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 on page 8-42], except the model in [6 on page 8-42] 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
```

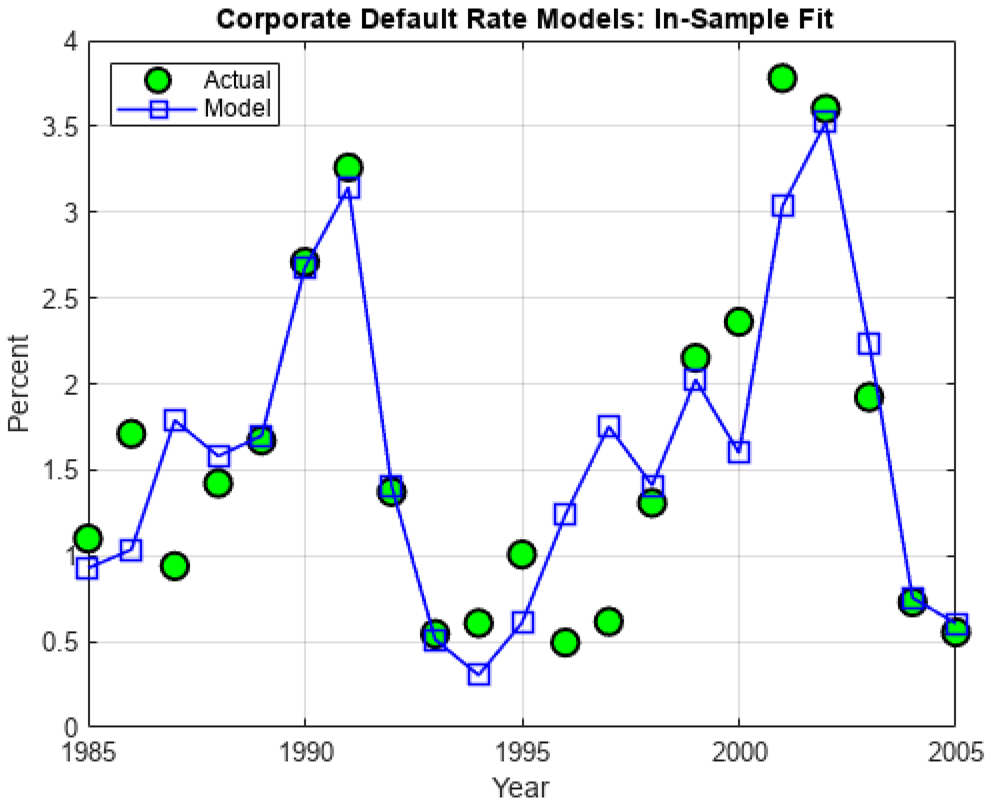
```
fprintf('%1.2f  %1.2f %1.2f  %1.2f   %1.4f\n',...
   [stats.beta;stats.adjrsquare])
```

```
-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.

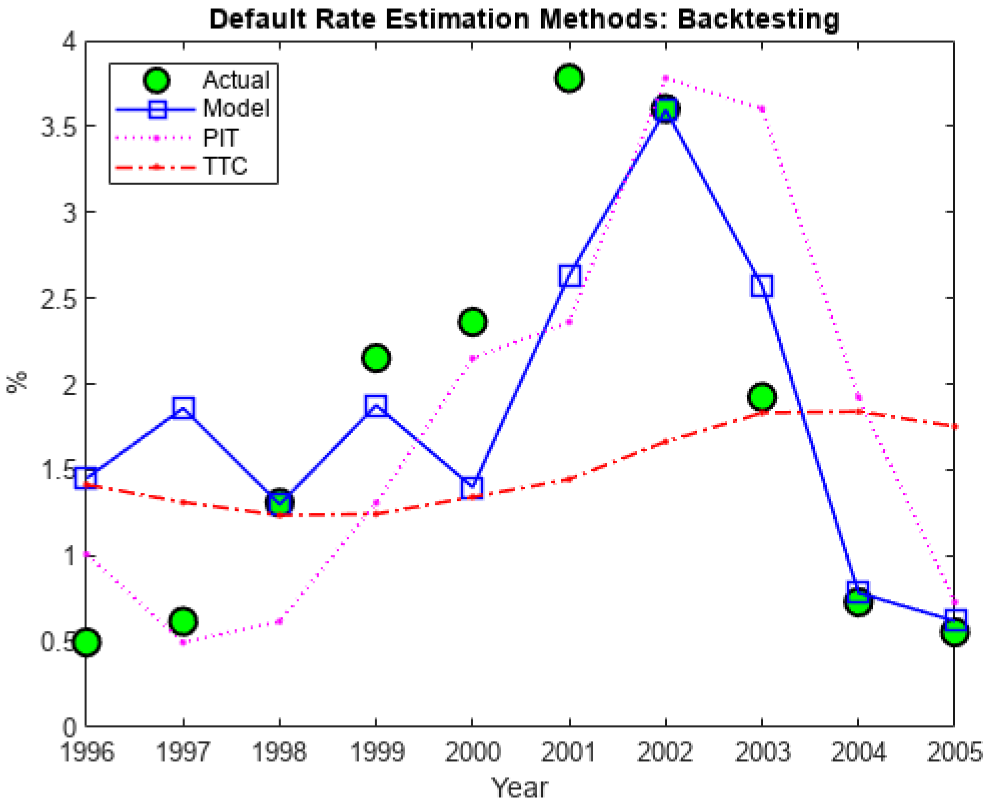
For backtesting, relative performance of a model, when compared to alternatives, is easier to assess than the performance of a model in isolation. Here we include two alternatives to determine next year's default rate, both likely candidates in practice. One is the TTC default rate, estimated with data 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
```

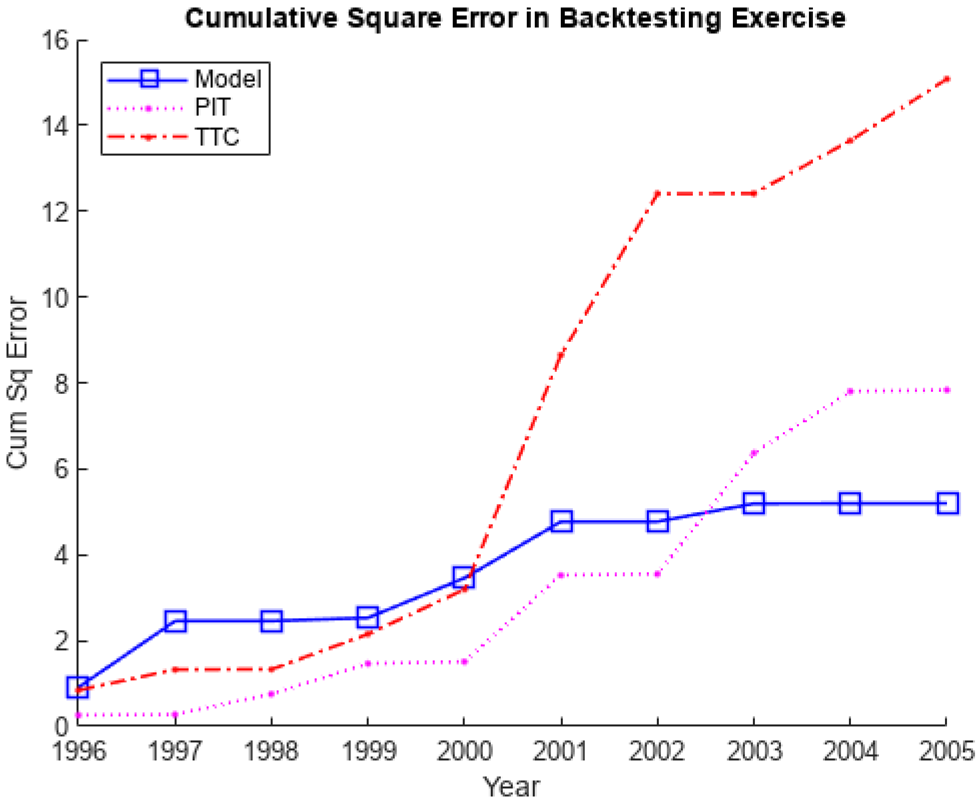
Here are the predictions of the three alternative approaches, compared to the actual default rates observed. Unsurprisingly, TTC shows a very poor predictive power. However, it is not obvious whether PIT or the linear regression model makes better predictions in this 10-year time span.

```
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

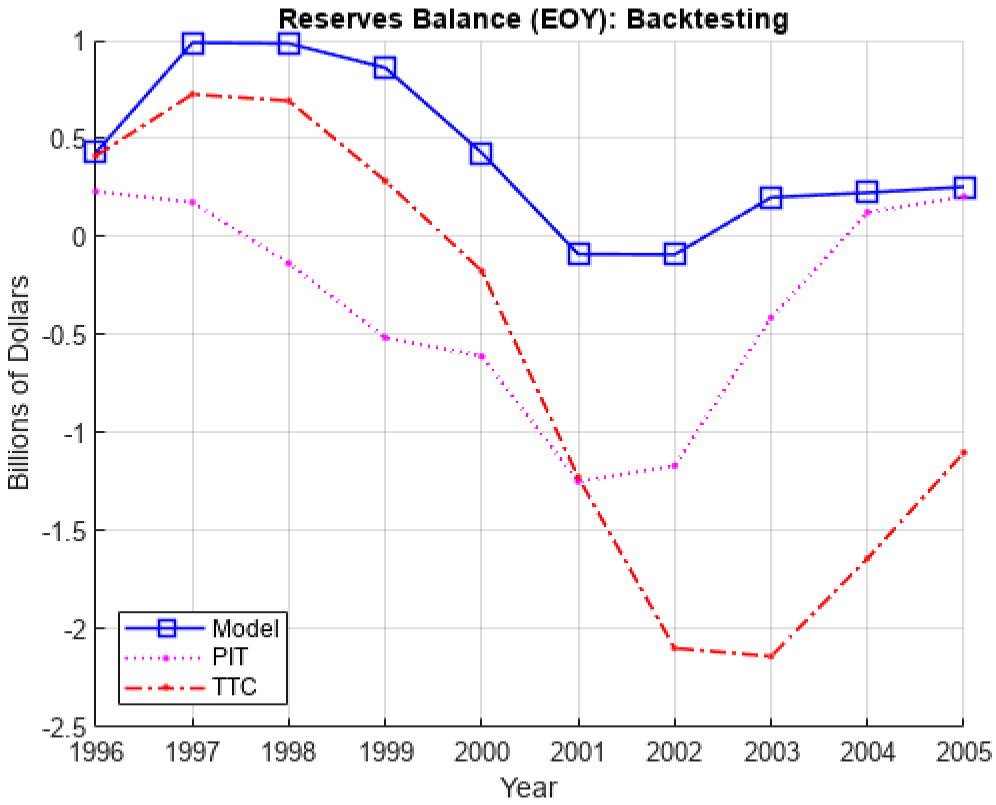
$$Reserves_t = Reserves_{t-1} + (EL_t - AL_t)$$

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.

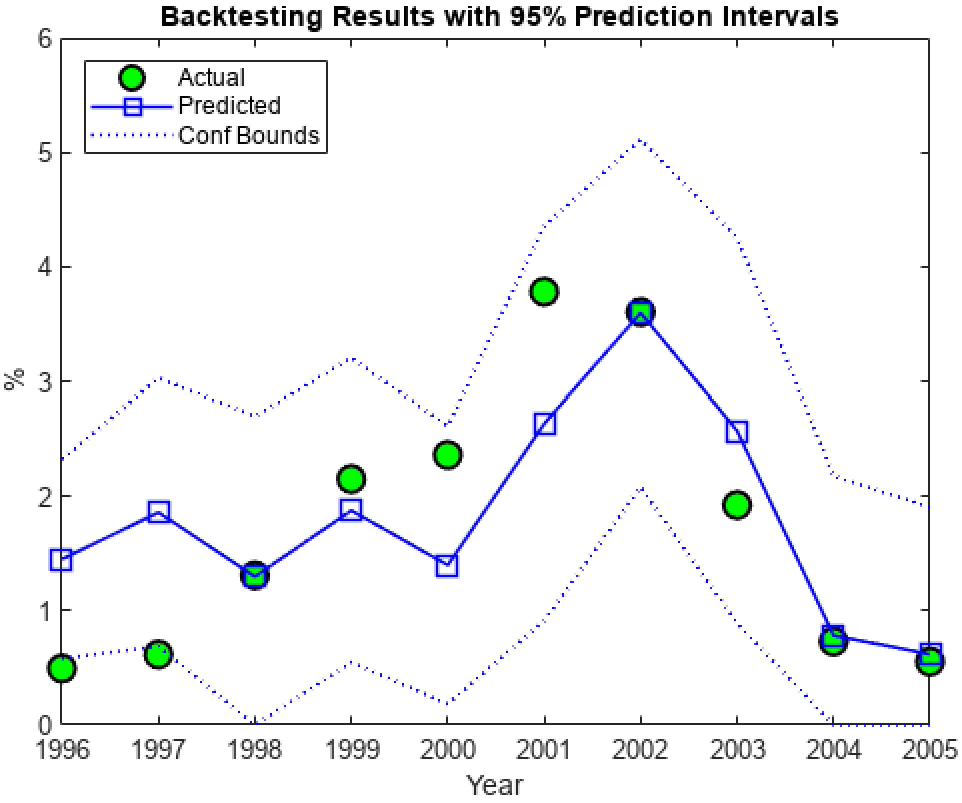
In contrast, both TTC and PIT reach a deficit of 1.2 billion by 2001. Things get worse for TTC in the next two years, reaching a deficit of 2.1 billion by 2003. PIT does make a correction quickly after 2001, and by 2004 the reserves have a surplus. Yet, both TTC and PIT lead to more deficit years than 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 on page 8-41]. 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%
```

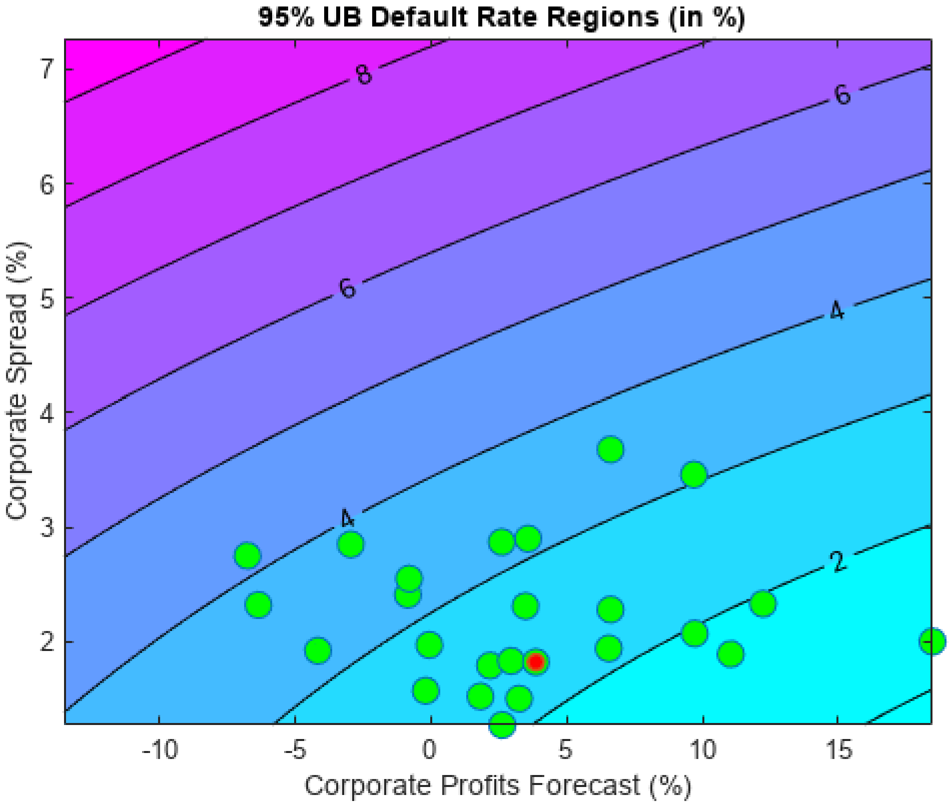
```
fprintf('\nCorrelation between CPF and SPR: %4.3f\n',corr(CPF,SPR));
```

```
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 %)}'])
```



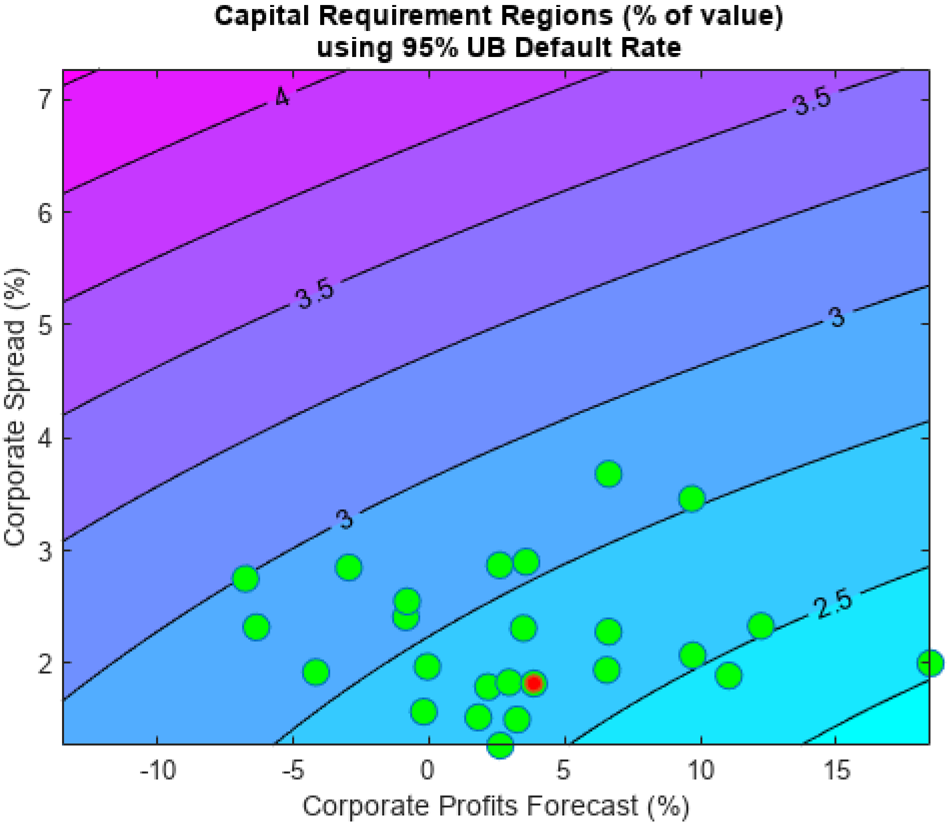
Very different predictor values result in similar default rate levels. For example, consider a profits forecast around 10% with a spread of 3.5%, and a profits forecast of -2.5% with a spread of 2%, they both result in a default rate slightly above 3%. Also, only one point in the available history yields a default rate higher than 4%.

Monetary terms, once again, may be more meaningful. We use the Basel II capital requirements formula (see [2 on page 8-41]) to translate the default rates into a monetary measure. The Basel II 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 the Basel II 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® Parallel 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 on page 8-42], for example, a single parameter related to the proportion of downgrades is used, and both [6 on page 8-42] and [8 on page 8-42] 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).

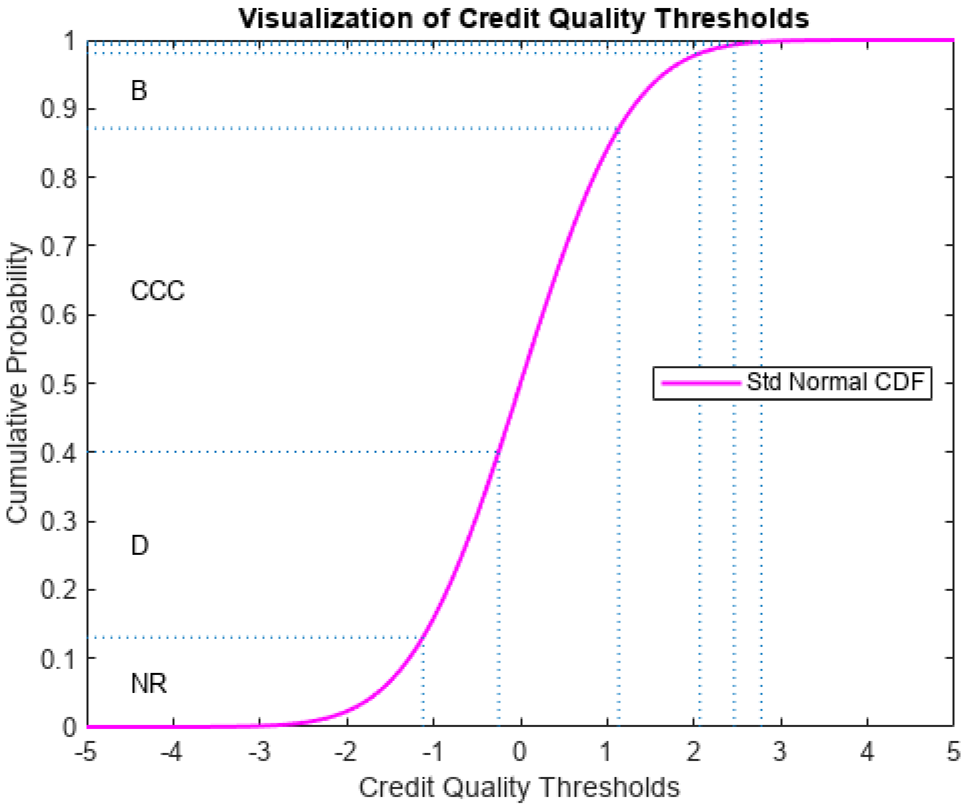
```
thresholdMat = transprobtothresholds(tmTTC);
Example_DisplayTransitions(thresholdMat,[],...
   {'AAA','AA','A','BBB','BB','B','CCC'},...
   {'AAA','AA','A','BBB','BB','B','CCC','D','NR'})
```

```
          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
```

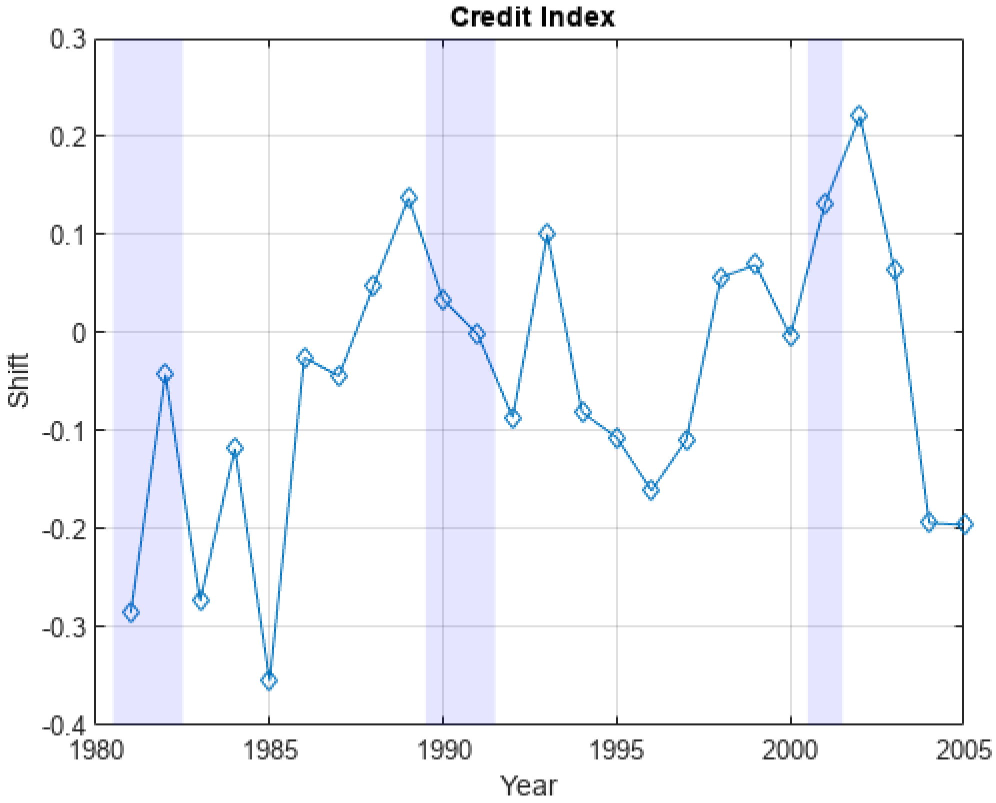
Given a particular PIT matrix, the idea in [6 on page 8-42] and [8 on page 8-42] is to vary the shifting parameter applied to the TTC thresholds so that the resulting transition matrix is as close as possible to the PIT matrix. Closeness is measured as the sum of squares of differences between corresponding 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: https://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: https://www.bis.org/publ/bcbs155.htm.

[4] FRED, St. Louis Federal Reserve, Federal Reserve Economic Database, https://fred.stlouisfed.org/.

[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, https://www.nber.org/research/business-cycle-dating.

[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, https://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

`transprob` | `transprobprep` | `transprobbytotals` | `bootstrp` | `transprobgrouptotals` | `transprobtothresholds` | `transprobfromthresholds`

Related Examples

- “Credit Quality Thresholds” on page 8-43
- “Credit Rating by Bagging Decision Trees”

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:

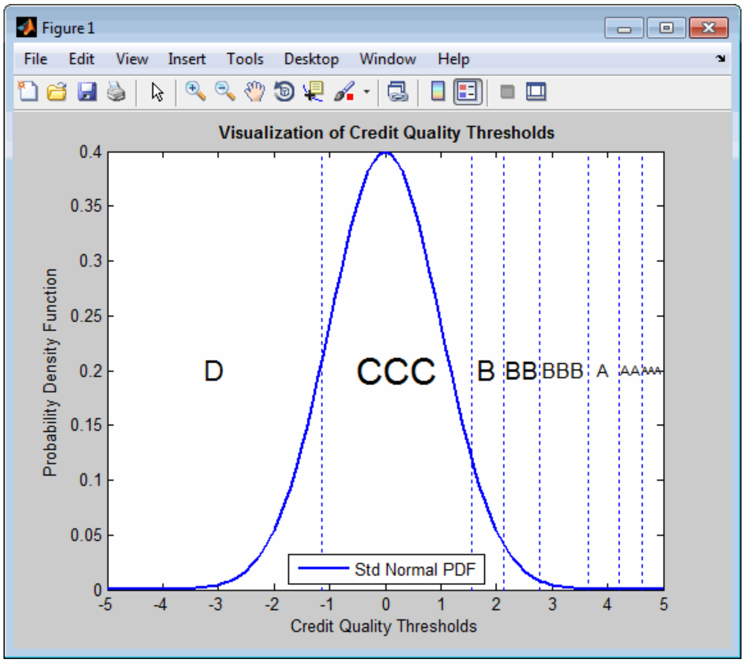
```
trans1 = transprobfromthresholds(thresh)
```

```
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

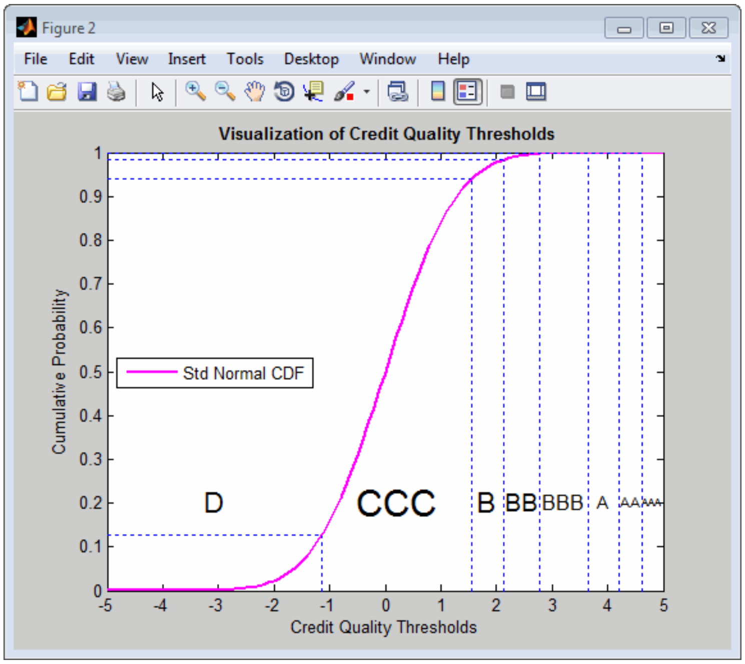
You can graphically represent the relationship between credit quality thresholds and transition probabilities. Here, this example shows the relationship for the `'CCC'` credit rating. In the plot, the thresholds are marked by the vertical lines and the transition probabilities are the area below the 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

`transprob` | `transprobprep` | `transprobbytotals` | `bootstrp` | `transprobgrouptotals` | `transprobtothresholds` | `transprobfromthresholds`

Related Examples

- “Estimation of Transition Probabilities” on page 8-2
- “Credit Rating by Bagging Decision Trees”
- “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

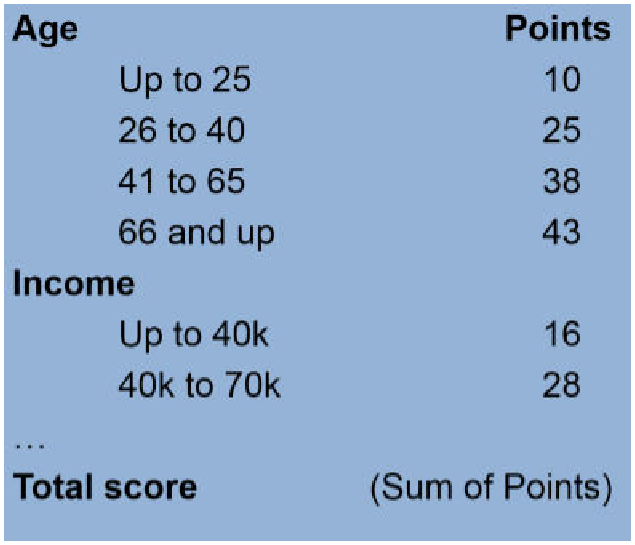
In this section...

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, and so on), 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.

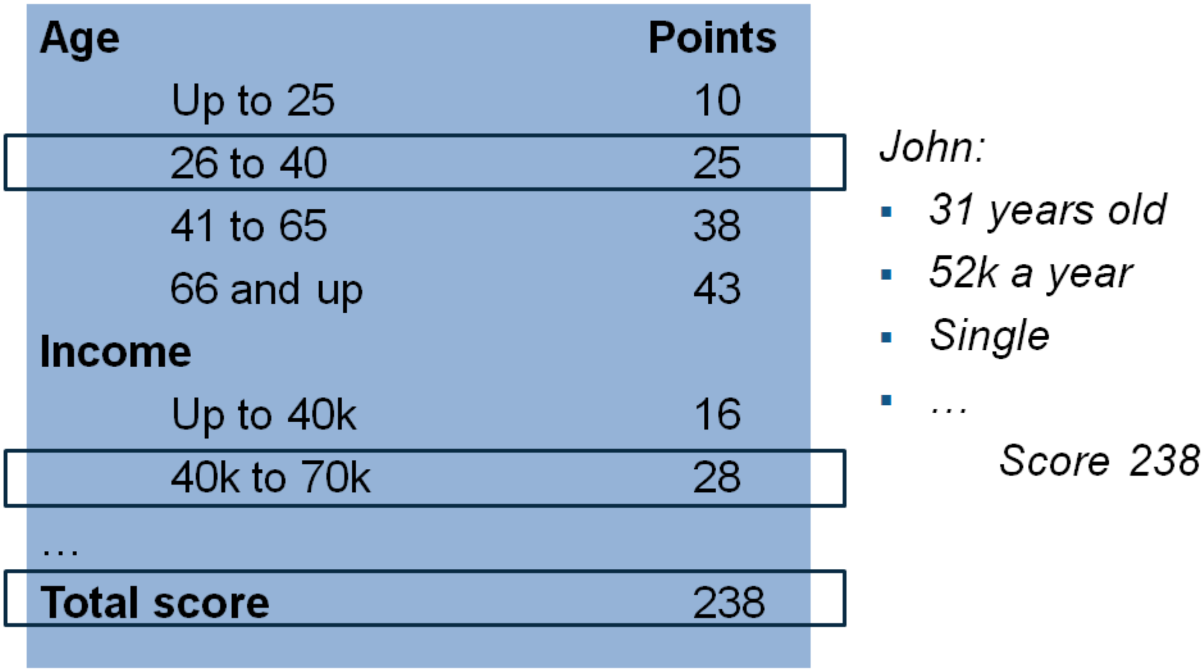
A credit scorecard is a lookup table that maps specific characteristics of a borrower into points. The total number of points becomes the credit score. Credit scorecards are a widely used type of credit scoring model. As such, the goal of a credit scorecard is to distinguish between customers who repay 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).



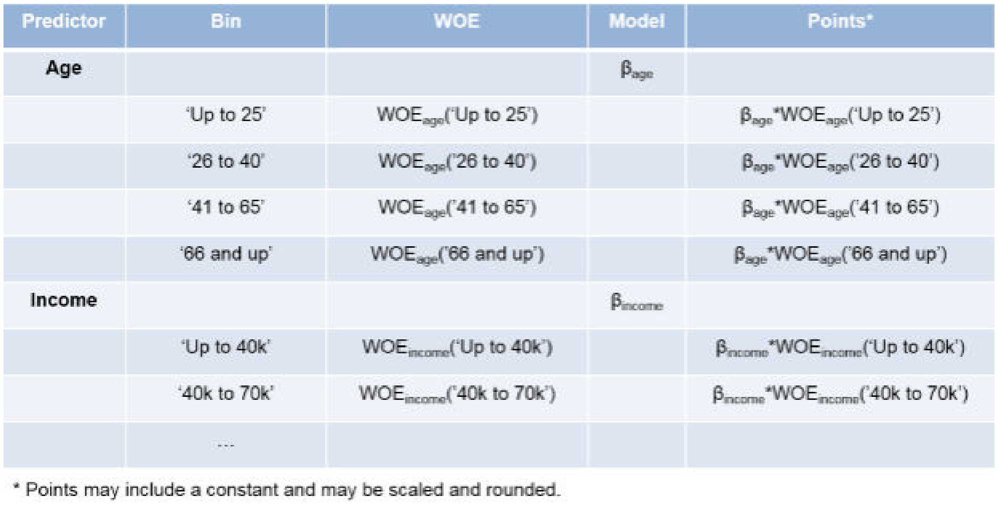
Age		Points
	Up to 25	10
	26 to 40	25
	41 to 65	38
	66 and up	43
Income		
	Up to 40k	16
	40k to 70k	28
…		
Total score		238

John:

- *31 years old*
- *52k a year*
- *Single*
- …

Score 238

Technically, to determine the credit scorecard points, 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

`creditscorecard` | `autobinning` | `bininfo` | `predictorinfo` | `modifypredictor` | `modifybins` | `bindata` | `plotbins` | `fitmodel` | `displaypoints` | `formatpoints` | `score` | `setmodel` | `probdefault` | `validatemodel`

Related Examples

- "Troubleshooting Credit Scorecard Results" on page 8-63

- “Case Study for Credit Scorecard Analysis” on page 8-70
- “Explore Fairness Metrics for Credit Scoring Model” (Risk Management Toolbox)

More About

- “Credit Scorecard Modeling Workflow” on page 8-51
- “Credit Scorecard Modeling Using Observation Weights” on page 8-54

Credit Scorecard Modeling Workflow

Create, model, and analyze credit scorecards as follows.

1. Use `screenpredictors` from Risk Management Toolbox™ to pare down a potentially large set of predictors to a subset that is most predictive of the credit score card response variable. Use this subset of predictors when creating the `creditscorecard` object. In addition, you can use **Threshold Predictors** to interactively set credit scorecard predictor thresholds using the output from `screenpredictors`.

   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 8-47) 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 Credit Scorecard Analysis” on page 8-70.
2. Create a `creditscorecard` object using training data.

   When you create a `creditscorecard` object for credit scorecard, you can specify “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 8-47) 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 Credit Scorecard Analysis” on page 8-70.
3. 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 Credit Scorecard Analysis” on page 8-70.
4. 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 Credit Scorecard Analysis” on page 8-70.

Alternatively, you can apply equality, inequality, or bound constraints to fit a logistic regression model to the WOE data from the `creditscorecard` object using `fitConstrainedModel`.

5 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 Credit Scorecard Analysis” on page 8-70.

Optionally, you can create a compact credit scorecard using

To create a `compactCreditScorecard` object, use `compact` to create a `compactCreditScorecard` object. You can then use the following functions `displaypoints`, `score`, and `probdefault` from the Risk Management Toolbox with the `compactCreditScorecard` object..

6 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 Credit Scorecard Analysis” on page 8-70.

7 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 Credit Scorecard Analysis” on page 8-70.

8 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 Credit Scorecard Analysis” on page 8-70.

For an example of this workflow, see “Case Study for Credit Scorecard Analysis” on page 8-70.

See Also

`creditscorecard` | `autobinning` | `bininfo` | `predictorinfo` | `modifypredictor` | `modifybins` | `bindata` | `plotbins` | `fitmodel` | `displaypoints` | `formatpoints` | `score` | `setmodel` | `probdefault` | `validatemodel`

Related Examples

- “Troubleshooting Credit Scorecard Results” on page 8-63
- “Case Study for Credit Scorecard Analysis” on page 8-70

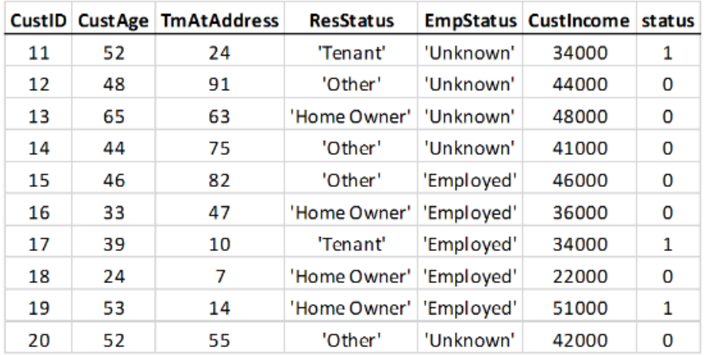
More About

- “About Credit Scorecards” on page 8-47
- “Credit Scorecard Modeling Using Observation Weights” on page 8-54
- “Credit Scorecard Modeling with Missing Values” on page 8-56

Credit Scorecard Modeling Using Observation Weights

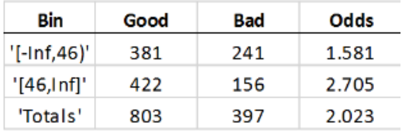
When creating a `creditscorecard` object, the table used for the input `data` argument either defines or does not define observational weights. If the `data` does not use weights, then the "counts" for `Good`, `Bad`, and `Odds` are used by credit score card functions. However, if the optional `WeightsVar` argument is specified 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

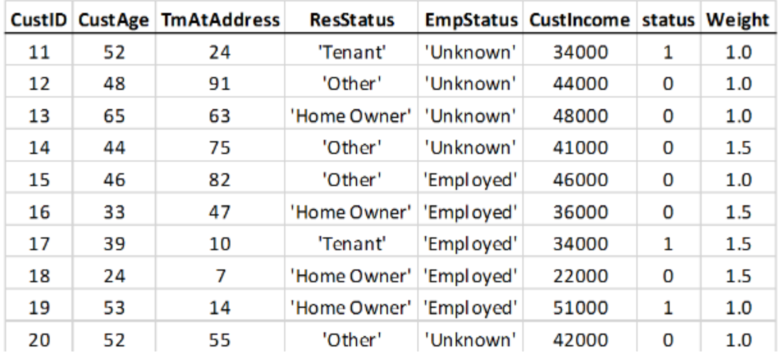
If you bin the customer age predictor data, with customers up to 45 years old in one bin, and 46 and up in another bin, you get these statistics:



Bin	Good	Bad	Odds
'[-Inf,46)'	381	241	1.581
'[46,Inf]'	422	156	2.705
'Totals'	803	397	2.023

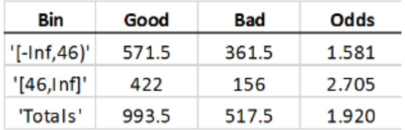
`Good` means the total number of rows with a `0` value in the `status` response variable. `Bad` the number of `1`’s in the `status` column. `Odds` is the ratio of `Good` to `Bad`. The `Good`, `Bad`, and `Odds` is reported for each bin. This means that there are 381 people in the sample who are 45 and 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

`creditscorecard` | `autobinning` | `bininfo` | `fitmodel` | `validatemodel`

More About

- “About Credit Scorecards” on page 8-47
- “Credit Scorecard Modeling Workflow” on page 8-51

Credit Scorecard Modeling with Missing Values

This example shows how to handle missing values when you work with creditscorecard objects. First, the example shows how to use the creditscorecard functionality to create an explicit bin for missing data with corresponding points. Then, this example describes four different ways to "treat" the missing data on page 8-61 to get a final credit scorecard with no explicit bins for missing values.

Develop a Credit Scorecard with Explicit Bins for Missing Values

When you create a creditscorecard object, the data can contain missing values. When using creditscorecard to create a creditscorecard object, you can set the name-value pair argument for 'BinMissingData' set to true. In this case, the missing data for numeric predictors (NaN values) and for categorical predictors (<undefined> values) is binned in a separate bin labeled <missing> that appears at the end of the bins. Predictors with no missing values in the training data have no <missing> bin. If you do not specify the 'BinMissingData' argument or if you set 'BinMissingData' to false, the creditscorecard function discards missing observations when computing frequencies of Good and Bad, and neither the bininfo nor plotbins functions reports such observations.

The <missing> bin remains in place throughout the scorecard modeling process. The final scorecard explicitly indicates the points to be assigned to missing values for predictors that have a <missing> bin. These points are determined from the weight-of-evidence (WOE) value of the <missing> bin and the predictor's coefficient in the logistic model. For predictors without an explicit <missing> bin, you can assign points to missing values using the name-value pair argument 'Missing' in formatpoints, as described in this example, or by using one of the four different ways to "treat" the missing data on page 8-61.

The dataMissing table in the CreditCardData.mat file has two predictors with missing values — CustAge and ResStatus, .

```
load CreditCardData.mat
head(dataMissing,5)
```

```
    CustID     CustAge    TmAtAddress      ResStatus       EmpStatus     CustIncome     TmWBank     Othe
    ______     _______    ___________     ___________     _________     __________     _______     ____

      1          53           62          <undefined>     Unknown         50000          55          Ye
      2          61           22          Home Owner      Employed        52000          25          Ye
      3          47           30          Tenant          Employed        37000          61          No
      4         NaN           75          Home Owner      Employed        53000          20          Ye
      5          68           56          Home Owner      Employed        53000          14          Ye
```

Create a creditscorecard object using the CreditCardData.mat file to load the dataMissing table with missing values. Set the 'BinMissingData' argument to true. Apply automatic binning.

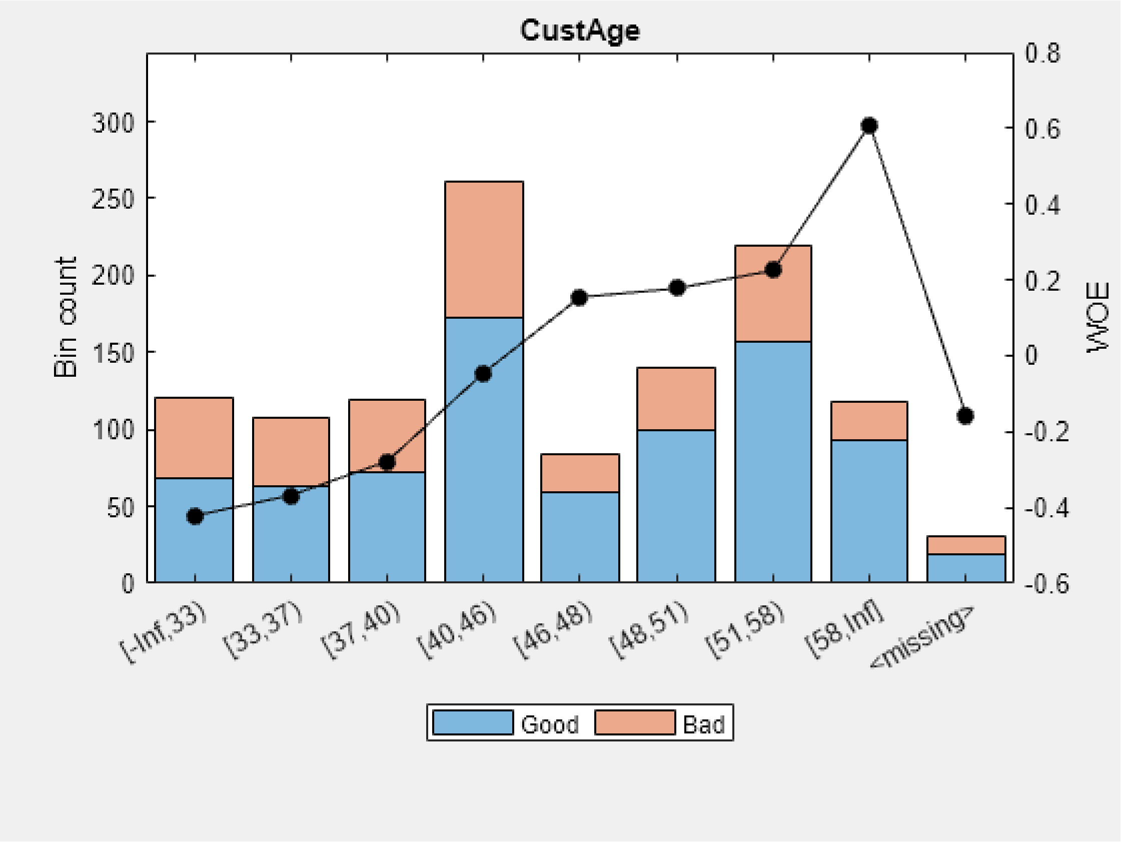
```
sc = creditscorecard(dataMissing,'IDVar','CustID','BinMissingData',true);
sc = autobinning(sc);
```

The bin information and bin plots for the predictors that have missing data both show a <missing> bin at the end.

```
bi = bininfo(sc,'CustAge');
disp(bi)
```

```
         Bin           Good    Bad     Odds       WOE        InfoValue
    _____________      ____    ___    ______    _________    __________
    {'[-Inf,33)'}     69       52      1.3269    -0.42156       0.018993
    {'[33,37)'  }     63       45         1.4    -0.36795       0.012839
    {'[37,40)'  }     72       47      1.5319     -0.2779      0.0079824
    {'[40,46)'  }    172       89      1.9326    -0.04556      0.0004549
    {'[46,48)'  }     59       25        2.36     0.15424      0.0016199
    {'[48,51)'  }     99       41      2.4146     0.17713      0.0035449
    {'[51,58)'  }    157       62      2.5323     0.22469      0.0088407
    {'[58,Inf]' }     93       25        3.72     0.60931       0.032198
    {'<missing>'}     19       11      1.7273    -0.15787     0.00063885
    {'Totals'   }    803      397      2.0227         NaN       0.087112
```

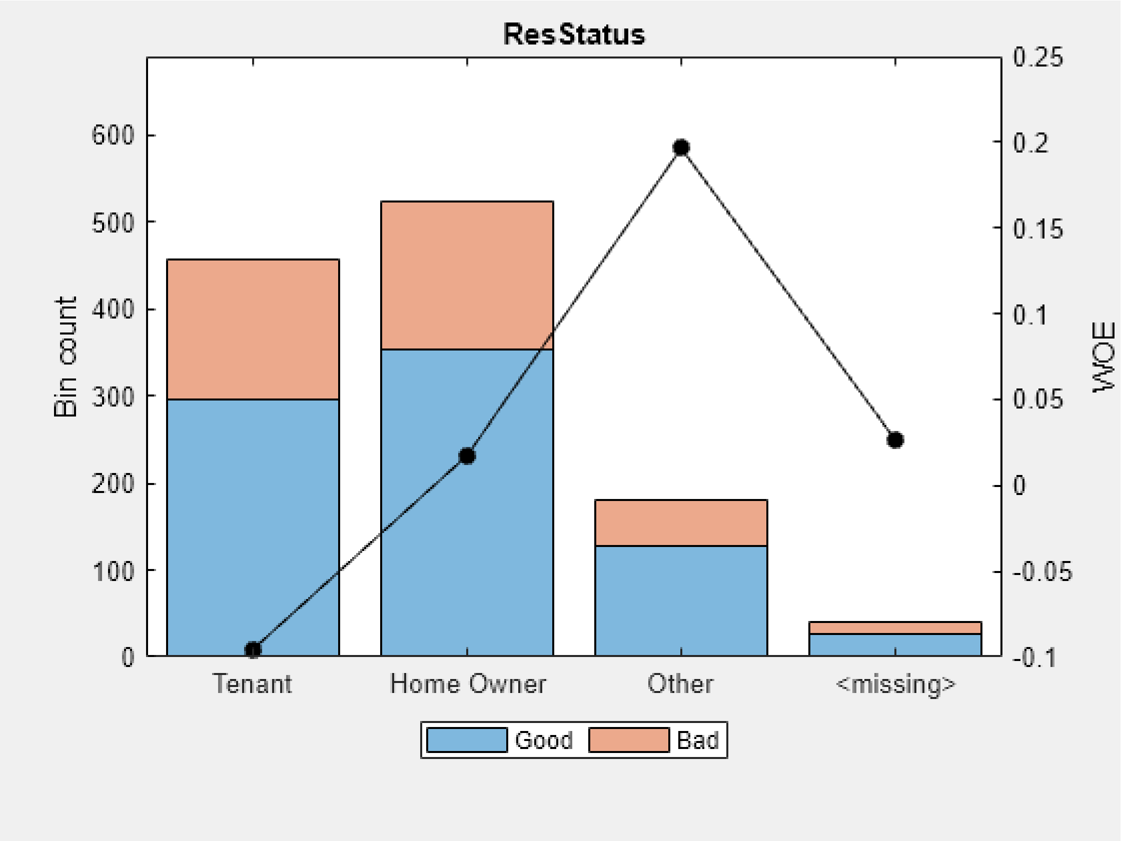
```
plotbins(sc,'CustAge')
```



```
bi = bininfo(sc,'ResStatus');
disp(bi)
```

```
          Bin             Good    Bad     Odds        WOE         InfoValue 
    ________________    ____    ___    ______    _________    __________

    {'Tenant'     }     296     161    1.8385    -0.095463     0.0035249
    {'Home Owner'}      352     171    2.0585     0.017549    0.00013382
    {'Other'      }     128      52    2.4615      0.19637     0.0055808
    {'<missing>' }       27      13    2.0769     0.026469    2.3248e-05
    {'Totals'     }     803     397    2.0227          NaN     0.0092627
```

```
plotbins(sc,'ResStatus')
```



The training data for the `'CustAge'` and `'ResStatus'` predictors has missing data (`NaN`s and `<undefined>`). The binning process estimates WOE values of `-0.15787` and `0.026469`, respectively, for the missing data in these predictors.

The training data for `EmpStatus` and `CustIncome` has no explicit bin for `<missing>` values because there are no missing values for these predictors.

```
bi = bininfo(sc,'EmpStatus');
disp(bi)
```

```
        Bin          Good    Bad     Odds        WOE       InfoValue
    _____________    ____    ___    ______    ________    _________

    {'Unknown' }     396     239    1.6569    -0.19947    0.021715
    {'Employed'}     407     158    2.5759      0.2418    0.026323
    {'Totals'  }     803     397    2.0227         NaN    0.048038
```

```
bi = bininfo(sc,'CustIncome');
disp(bi)
```

```
          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 `fitmodel` to fit a logistic regression model using WOE values. `fitmodel` internally transforms all the predictor variables into WOE values, using the bins found during the automatic binning process. By default, `fitmodel` then fits a logistic regression model using a stepwise method. For predictors that have missing data, there is an explicit `<missing>` bin with a corresponding WOE value computed from the data. When you use `fitmodel`, the corresponding WOE value for the `<missing>` bin is applied when the function performs the WOE transformation.

```
[sc,mdl] = fitmodel(sc,'display','off');
```

Scale the scorecard points by the points-to-double-the-odds (PDO) method using the `'PointsOddsAndPDO'` argument of `formatpoints`. 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).

Display the scorecard showing the scaled points for predictors retained in the fitting model.

```
sc = formatpoints(sc,'PointsOddsAndPDO',[500 2 50]);
PointsInfo = displaypoints(sc)
```

```
PointsInfo=38×3 table
     Predictors           Bin           Points
    _____________    ______________    ______

    {'CustAge'  }    {'[-Inf,33)' }    54.062
    {'CustAge'  }    {'[33,37)'   }    56.282
    {'CustAge'  }    {'[37,40)'   }    60.012
    {'CustAge'  }    {'[40,46)'   }    69.636
    {'CustAge'  }    {'[46,48)'   }    77.912
    {'CustAge'  }    {'[48,51)'   }     78.86
    {'CustAge'  }    {'[51,58)'   }     80.83
    {'CustAge'  }    {'[58,Inf]'  }     96.76
    {'CustAge'  }    {'<missing>' }    64.984
    {'ResStatus'}    {'Tenant'    }    62.138
    {'ResStatus'}    {'Home Owner'}    73.248
    {'ResStatus'}    {'Other'     }    90.828
    {'ResStatus'}    {'<missing>' }    74.125
    {'EmpStatus'}    {'Unknown'   }    58.807
    {'EmpStatus'}    {'Employed'  }    86.937
    {'EmpStatus'}    {'<missing>' }       NaN
      ⋮
```

Notice that points for the `<missing>` bins for `CustAge` and `ResStatus` are explicitly shown (as `64.9836` and `74.1250`, respectively). These points are computed from the WOE value for the `<missing>` bin and the logistic model coefficients.

Points for predictors that have no missing data in the training set, by default, are set to `NaN` and they lead to a score of `NaN` when you run `score`. This can be changed by updating the name-value pair argument `'Missing'` in `formatpoints` to indicate how to treat missing data for scoring purposes.

The scorecard is ready for scoring new data sets. You can also use the scorecard to compute probabilities of default or perform model validation. For details, see `score`, `probdefault`, and validatemodel. To further explore the handling of missing data, take a few rows from the original data as test data and introduce some missing data.

```
tdata = dataMissing(11:14,mdl.PredictorNames); % Keep only the predictors retained in the model
% Set some missing values
tdata.CustAge(1) = NaN;
tdata.ResStatus(2) = '<undefined>';
tdata.EmpStatus(3) = '<undefined>';
tdata.CustIncome(4) = NaN;
disp(tdata)
```

```
    CustAge      ResStatus      EmpStatus     CustIncome    TmWBank    OtherCC    AMBalance
    _______    ___________    ___________    __________    _______    _______    _________

      NaN      Tenant         Unknown          34000          44         Yes         119.8
       48      <undefined>    Unknown          44000          14         Yes        403.62
       65      Home Owner     <undefined>      48000           6         No         111.88
       44      Other          Unknown            NaN          35         No         436.41
```

Score the new data and see how points for missing data are differently assigned for CustAge and ResStatus and for EmpStatus and CustIncome. CustAge and ResStatus have an explicit <missing> bin for missing data. However, for EmpStatus and CustIncome, the score function sets the points to NaN.

```
[Scores,Points] = score(sc,tdata);
disp(Scores)
```

```
  481.2231
  520.8353
       NaN
       NaN
```

```
disp(Points)
```

```
    CustAge    ResStatus    EmpStatus    CustIncome    TmWBank    OtherCC    AMBalance
    _______    _________    _________    __________    _______    _______    _________

    64.984      62.138       58.807        67.893      61.858     75.622      89.922  
     78.86      74.125       58.807        82.439      61.061     75.622      89.922  
     96.76      73.248          NaN        96.969      51.132     50.914      89.922  
    69.636      90.828       58.807           NaN      61.858     50.914      89.922  
```

Use the name-value pair argument 'Missing' in formatpoints to choose how to assign points to missing values for predictors that do not have an explicit <missing> bin. For this example, use the 'MinPoints' option for the 'Missing' argument. For EmpStatus and CustIncome, the minimum numbers of points in the scorecard are 58.8072 and 29.3753, respectively. You can also treat missing values using one of the four different ways to "treat" the missing data on page 8-61.

```
sc = formatpoints(sc,'Missing','MinPoints');
[Scores,Points] = score(sc,tdata);
disp(Scores)
```

```
  481.2231
  520.8353
  517.7532
  451.3405
```

```
disp(Points)
```

```
    CustAge    ResStatus    EmpStatus    CustIncome    TmWBank    OtherCC    AMBalance
    _______    _________    _________    __________    _______    _______    _________

    64.984      62.138       58.807        67.893      61.858     75.622      89.922
     78.86      74.125       58.807        82.439      61.061     75.622      89.922
     96.76      73.248       58.807        96.969      51.132     50.914      89.922
    69.636      90.828       58.807        29.375      61.858     50.914      89.922
```

Four Approaches for Treating Missing Data and Developing a New Credit Scorecard

There are four different approaches for treating missing data.

Approach 1: Fill missing data using the `fillmissing` function of the `creditscorecard` object

The `creditscorecard` object supports a `fillmissing` function. When you call the function on a predictor or group of predictors, the `fillmissing` function fills the missing data with the user-specified statistic. `fillmissing` supports the fill values `'mean'`, `'median'`, `'mode'`, and `'constant'`, as well as the option to switch back to the original data.

The advantage of using `fillmissing` is that the `creditscorecard` object keeps track of the fill value and also applies it to the validation data. The limitation of this approach is that only basic statistics are used to fill missing data.

For more information on **Approach 1**, see `fillmissing`.

Approach 2: Fill missing data using the MATLAB® `fillmissing` function

MATLAB® supports a `fillmissing` function that you can use before creating a `creditscorecard` object to treat missing values in numeric and categorical data. The advantage of this method is that you can use all the options available in `fillmissing` to fill missing data, as well as other MATLAB functionality, such as `standardizeMissing` and features for the treatment of outliers. However, the downside is that you are responsible for the same transformations to the validation data before scoring as the `fillmissing` function is outside of the `creditscorecard` object.

For more information on **Approach 2**, see “Treat Missing Data in a Credit Scorecard Workflow Using MATLAB® fillmissing” on page 8-130.

Approach 3: Impute missing data using the k-nearest neighbors (KNN) algorithm

This KNN approach considers multiple predictors as compared to Approach 1 and Approach 2. Like **Approach 2**, the KNN approach is done outside the `creditscoreacrd` workflow, and consequently, you need to perform imputation for both the training and validation data.

For more information on **Approach 3**, see “Impute Missing Data in the Credit Scorecard Workflow Using the k-Nearest Neighbors Algorithm” on page 8-118.

Approach 4: Impute missing data using the random forest algorithm

This random forest approach is similar to **Approach 3** and uses multiple predictors to impute missing values. Because the approach is outside the `creditscorecard` workflow, you need to perform imputation for both the training and validation data.

For more information on **Approach 4**, see “Impute Missing Data in the Credit Scorecard Workflow Using the Random Forest Algorithm” on page 8-125.

See Also

`creditscorecard` | `bininfo` | `plotbins` | `fillmissing`

More About

- “About Credit Scorecards” on page 8-47
- “Credit Scorecard Modeling Workflow” on page 8-51

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 8-51.

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`.

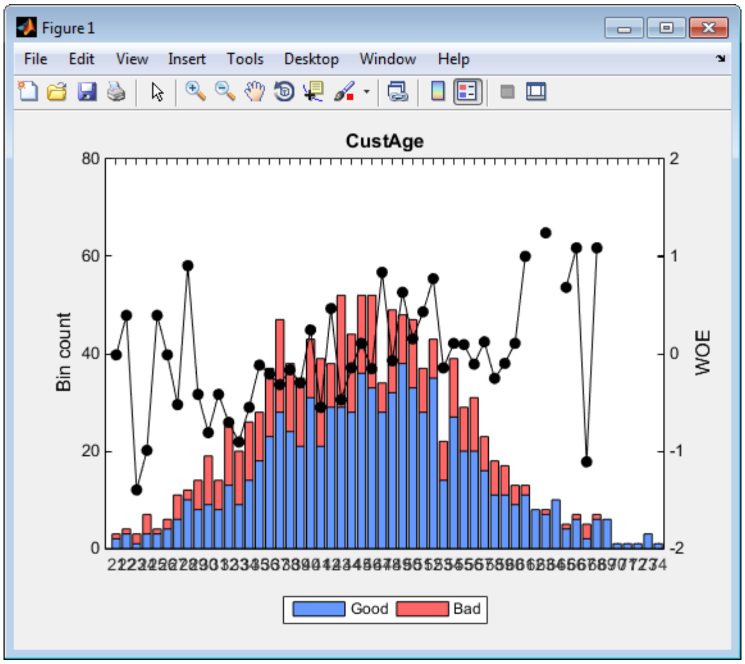
```
load CreditCardData
sc = creditscorecard(data,'IDVar','CustID','GoodLabel',0);
modifybins(sc,'CustIncome','CutPoints',[20 30 50 65]);
```

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
```

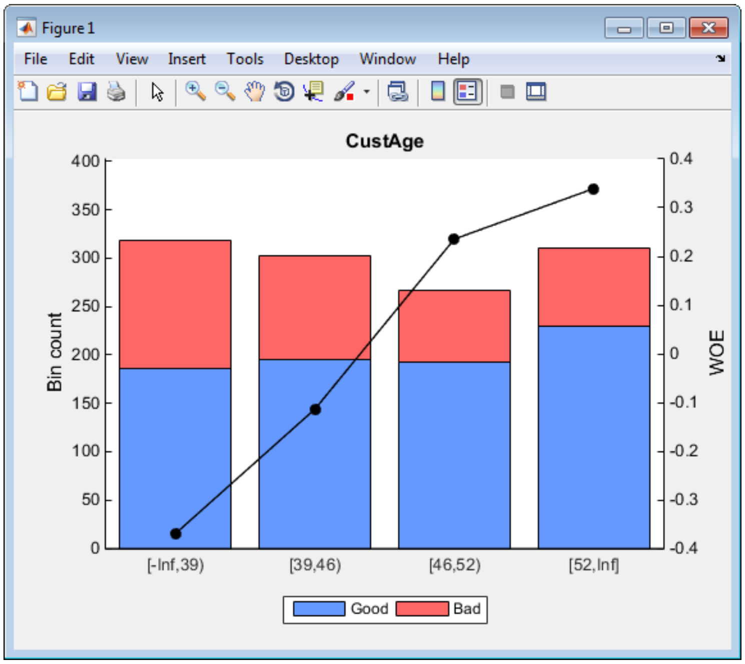


The plot for CustAge is not readable because it has too many bins. Also, bininfo returns data that have Inf values for the WOE due to zero observations for either **Good** or **Bad**.

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
```

Also, 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

`creditscorecard` | `autobinning` | `bininfo` | `predictorinfo` | `modifypredictor` | `modifybins` | `bindata` | `plotbins` | `fitmodel` | `displaypoints` | `formatpoints` | `score` | `setmodel` | `probdefault` | `validatemodel`

Related Examples

- “Case Study for Credit Scorecard Analysis” on page 8-70

More About

- “About Credit Scorecards” on page 8-47
- “Credit Scorecard Modeling Workflow” on page 8-51

Case Study for 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). If your `data` contains many predictors, you can first use `screenpredictors` (Risk Management Toolbox) to pare down a potentially large set of predictors to a subset that is most predictive of the credit scorecard response variable. You can then use this subset of predictors when creating the `creditscorecard` object. In addition, you can use Threshold Predictors (Risk Management Toolbox) to interactively set credit scorecard predictor thresholds using the output from `screenpredictors` (Risk Management Toolbox).

When creating a `creditscorecard` object, 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 or `'BinMissingData'` to bin missing data.

```
load CreditCardData
head(data)
```

```
    CustID    CustAge    TmAtAddress    ResStatus     EmpStatus    CustIncome    TmWBank    Othe
    ______    _______    ___________    __________    _________    __________    _______    ____

      1         53           62         Tenant        Unknown        50000         55         Yes
      2         61           22         Home Owner    Employed       52000         25         Yes
      3         47           30         Tenant        Employed       37000         61         No
      4         50           75         Home Owner    Employed       53000         20         Yes
      5         68           56         Home Owner    Employed       53000         14         Yes
      6         65           13         Home Owner    Employed       48000         59         Yes
      7         34           32         Home Owner    Unknown        32000         26         Yes
      8         50           57         Other         Employed       51000         33         No
```

The variables in `CreditCardData` are customer ID, customer age, time at current address, residential status, employment status, customer income, time with bank, other credit card, average monthly balance, utilization rate, and the default status (response).

```
sc = creditscorecard(data,'IDVar','CustID')
```

```
sc = 
  creditscorecard with properties:

                GoodLabel: 0
              ResponseVar: 'status'
               WeightsVar: ''
                 VarNames: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustI
        NumericPredictors: {'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBalance'  'Uti
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
```

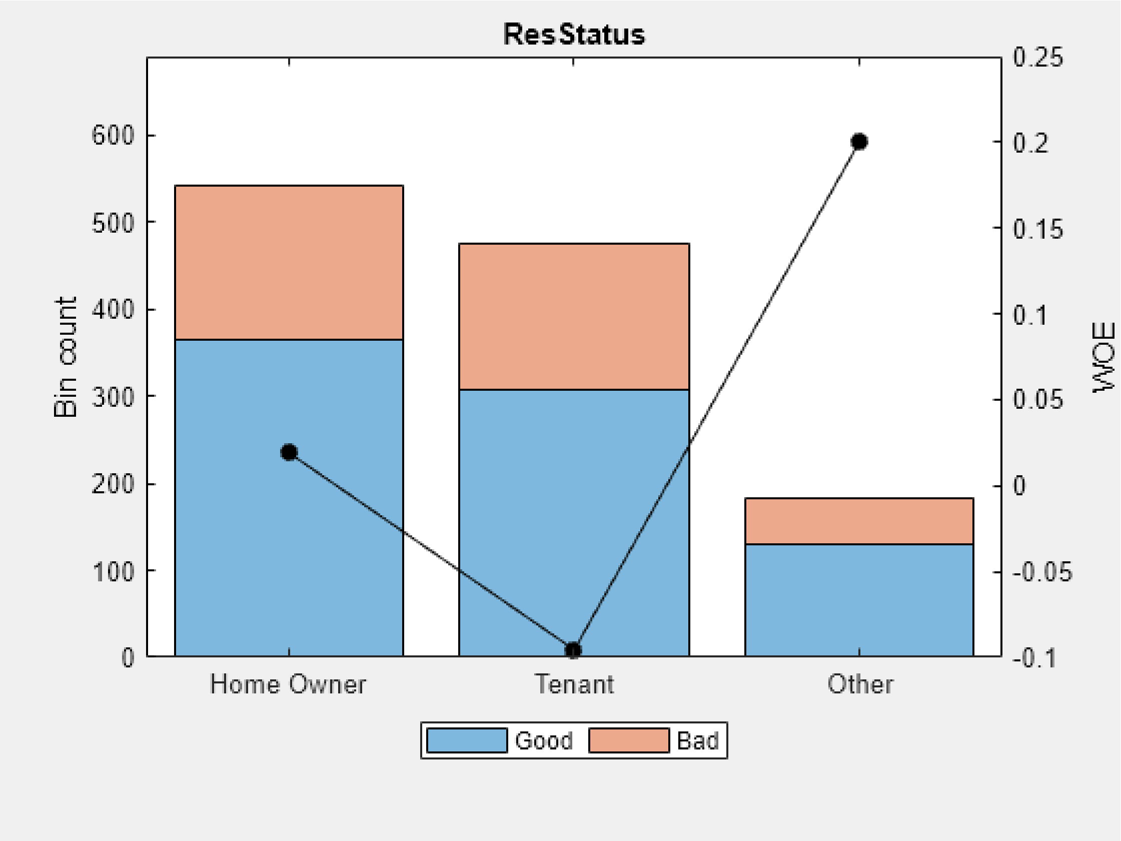
```
          BinMissingData: 0
                   IDVar: 'CustID'
           PredictorVars: {'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIncome'  'Tr
                    Data: [1200x11 table]
```

Perform some initial data exploration. Inquire about predictor statistics for the categorical variable `'ResStatus'` and plot the bin information for `'ResStatus'`.

```
bininfo(sc,'ResStatus')
```

```
ans=4×6 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
```

```
plotbins(sc,'ResStatus')
```



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'`.

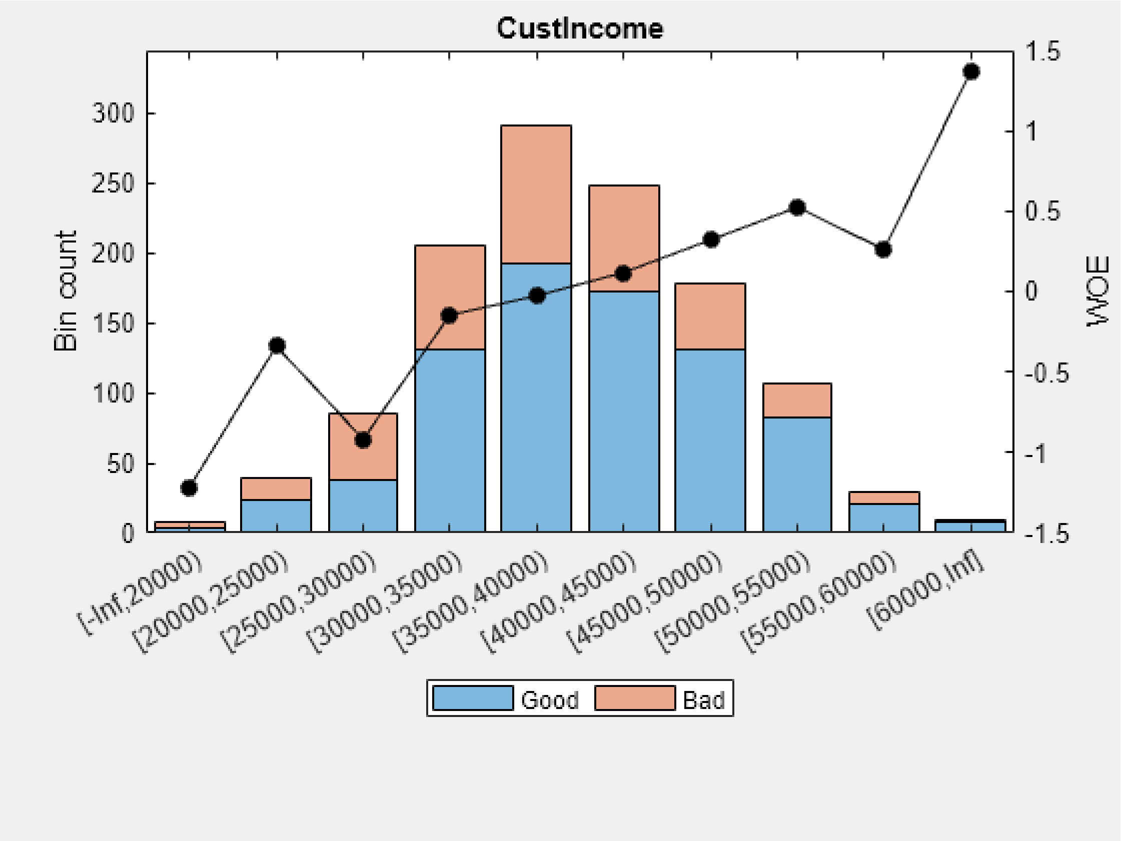
```
cp = 20000:5000:60000;
```

```
sc = modifybins(sc,'CustIncome','CutPoints',cp);
```

```
bininfo(sc,'CustIncome')
```

```
ans=11×6 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
```

```
plotbins(sc,'CustIncome')
```



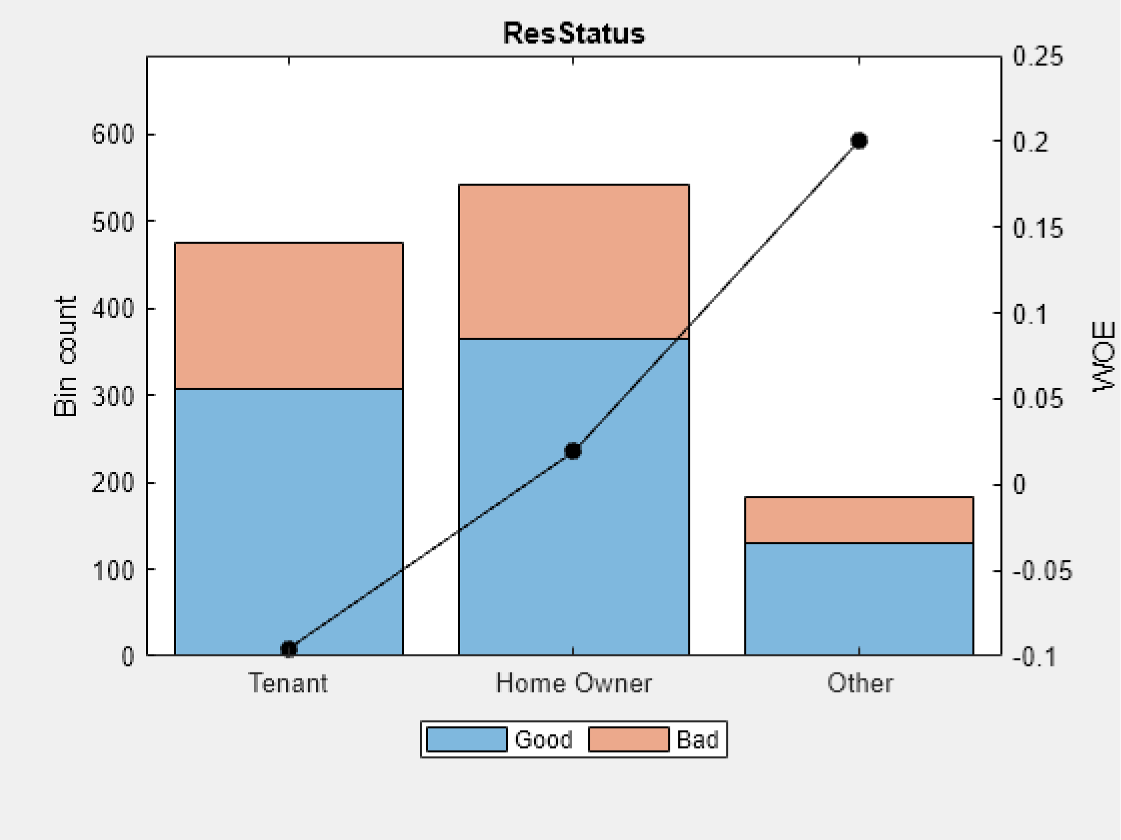
Step 2a. Automatically bin the data.

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`.

```
Predictor = ResStatus ▾;
plotbins(sc,Predictor)
```



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:

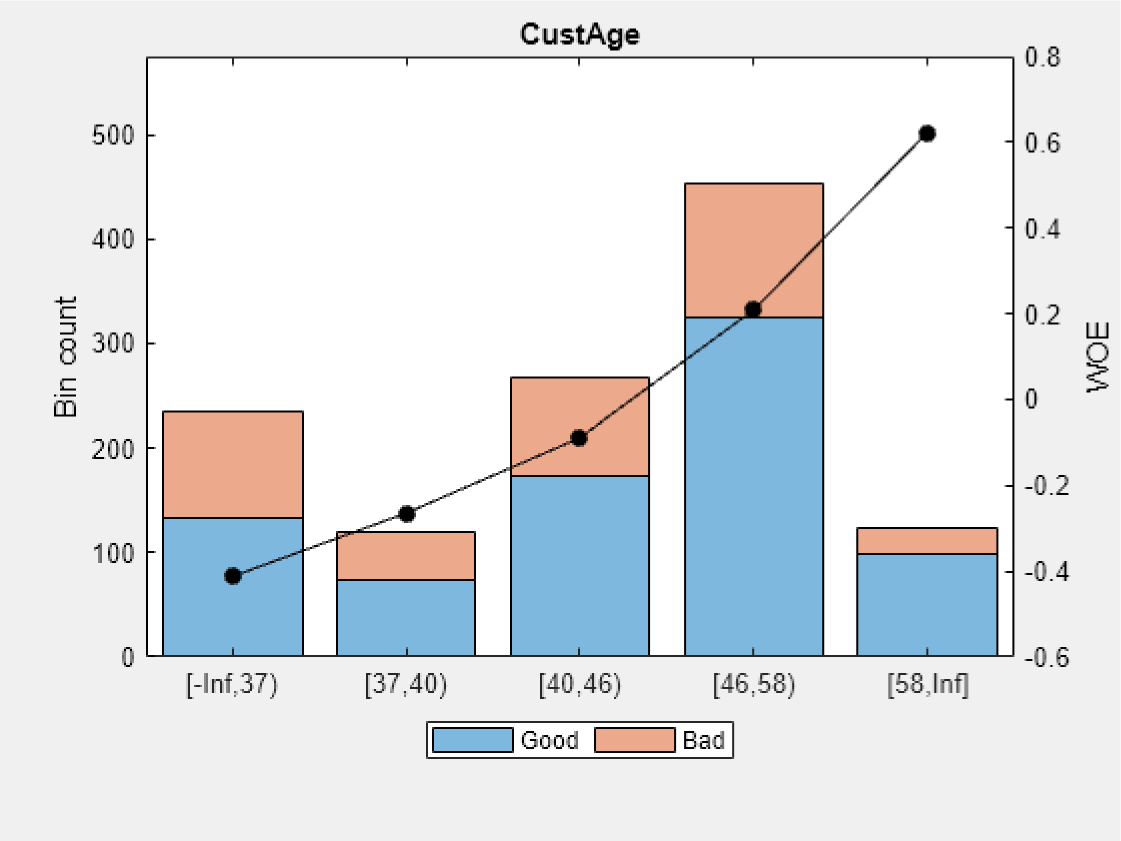
- Use the `bininfo` function with two output arguments where the second argument contains binning rules.
- Manually modify the binning rules using the second output argument from `bininfo`.
- Set the updated binning rules with `modifybins` and then use `plotbins` or `bininfo` to review the updated bins.

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:

```
Predictor = CustAge ;
[bi,cp] = bininfo(sc,Predictor)
```

```
bi=8×6 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


cp = 6×1

    33
    37
    40
    46
    48
    58
```

```
cp([1 5]) = []; % To merge bins 1 and 2, and bins 5 and 6
sc = modifybins(sc,'CustAge','CutPoints',cp);
plotbins(sc,'CustAge')
```



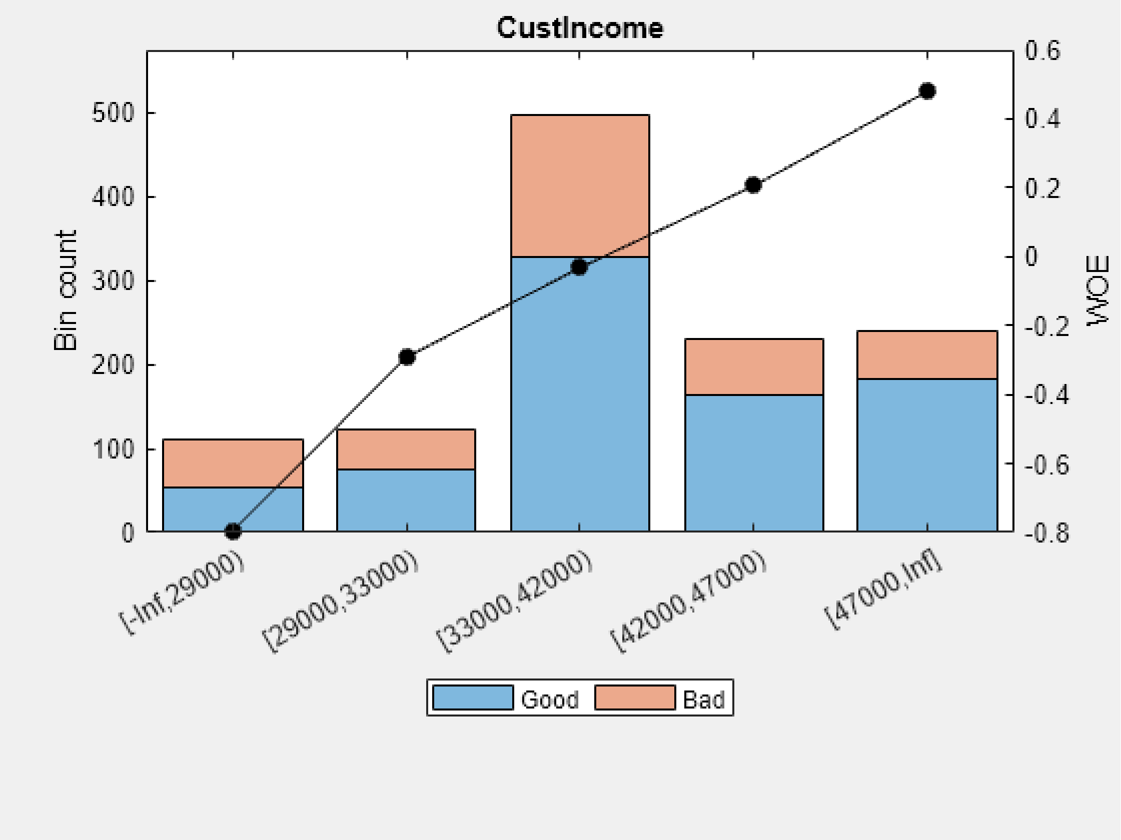
For `'CustIncome'`, based on the plot above, it is best to merge bins 3, 4 and 5 because they have similar WOE's. To merge these bins:

```
Predictor = CustIncome ▾;
[bi,cp] = bininfo(sc,Predictor)
```

```
bi=8×6 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 = 6×1

       29000
       33000
       35000
       40000
       42000
```

```
        47000
```

```
cp([3 4]) = []; % To merge bins 3, 4, and 5
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:

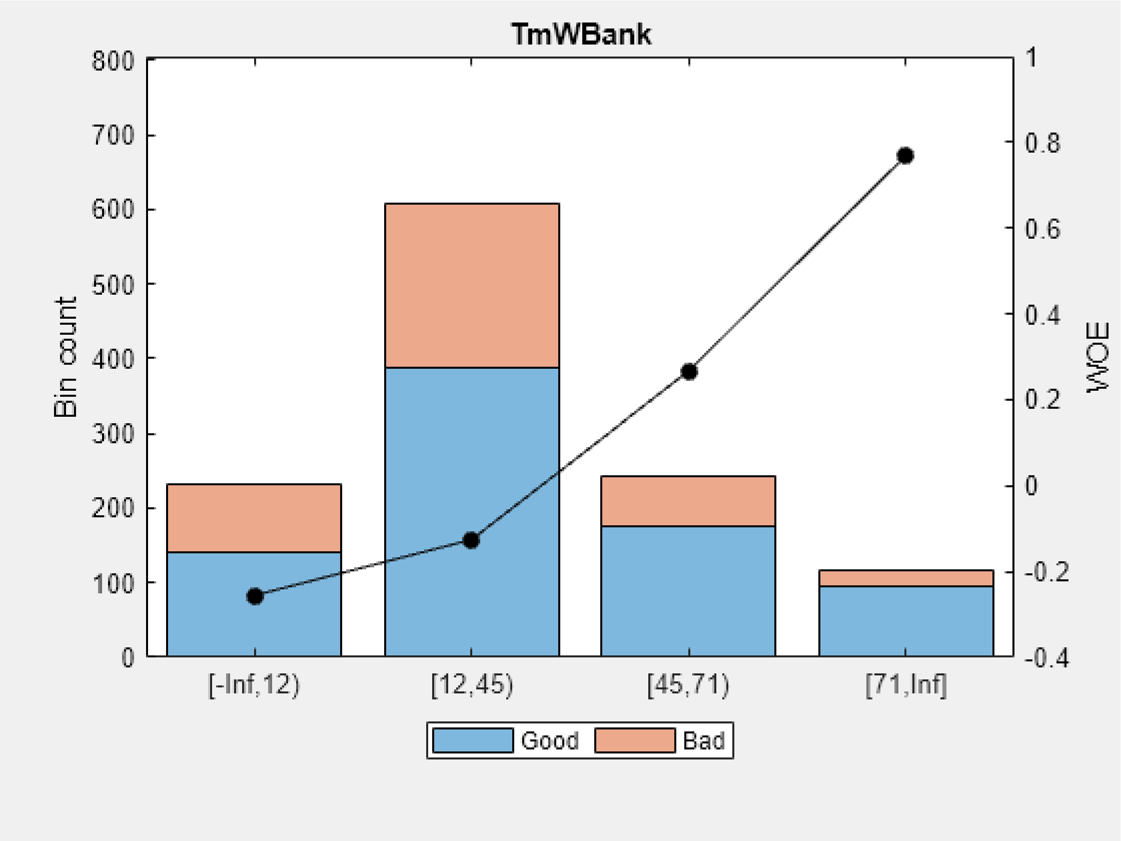
```
Predictor = TmWBank ;
[bi,cp] = bininfo(sc,Predictor)
```

```
bi=6×6 table
         Bin          Good    Bad      Odds        WOE       InfoValue
    _____________     ____    ___     ______    ________    _________

    {'[-Inf,12)'}     141      90     1.5667    -0.25547     0.013057
    {'[12,23)'  }     165      93     1.7742    -0.13107    0.0037719
    {'[23,45)'  }     224     125      1.792    -0.12109    0.0043479
    {'[45,71)'  }     177      67     2.6418     0.26704     0.013795
    {'[71,Inf]' }      96      22     4.3636     0.76889     0.049313
    {'Totals'   }     803     397     2.0227         NaN     0.084284
```

```
cp = 4×1
```

```
    12
    23
    45
    71
```

```
cp(2) = []; % To merge bins 2 and 3
sc = modifybins(sc,'TmWBank','CutPoints',cp);
plotbins(sc,'TmWBank')
```



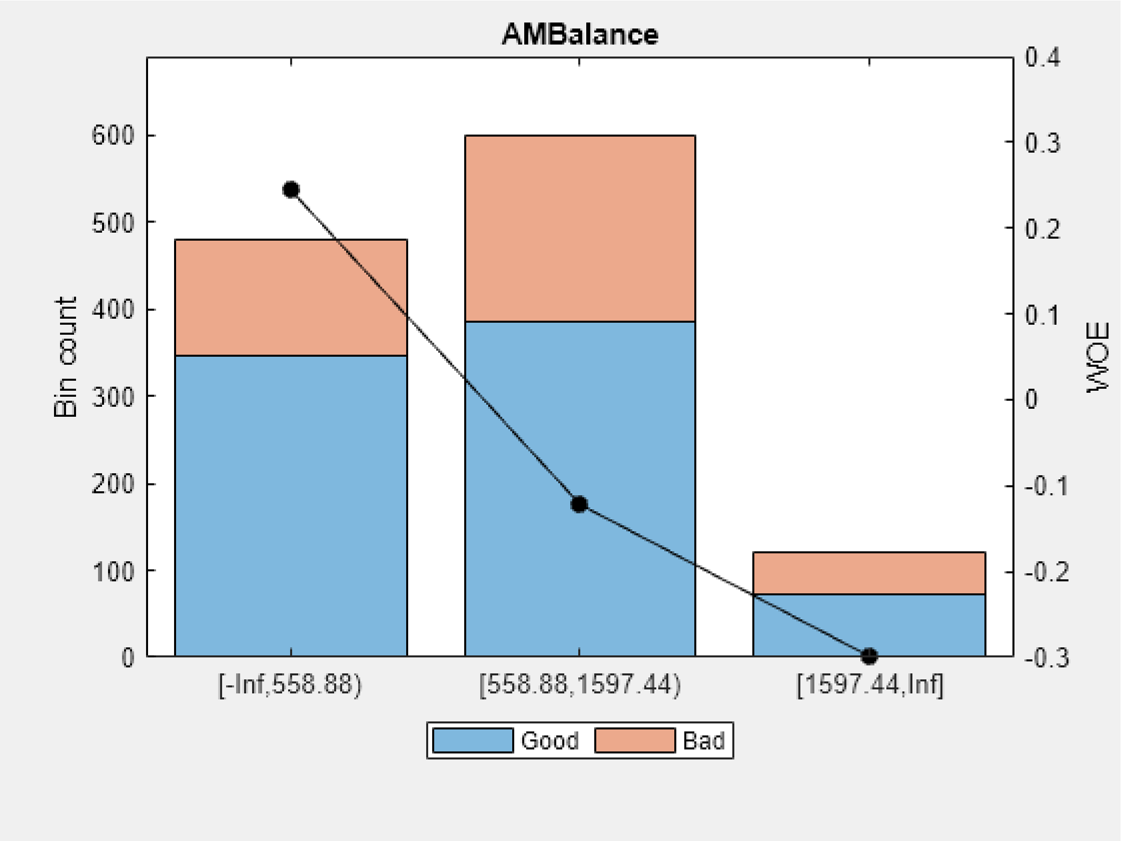
For `'AMBalance'`, based on the plot above, it is best to merge bins 2 and 3 because they have similar WOE's. To merge these bins:

```
Predictor = AMBalance ▾;
[bi,cp] = bininfo(sc,Predictor)
```

```
bi=5×6 table
             Bin              Good    Bad     Odds        WOE       InfoValue
    _____________________    ____    ___    ______    ________    _________

    {'[-Inf,558.88)'     }    346     134    2.5821     0.24418     0.022795
    {'[558.88,1254.28)'  }    309     171     1.807    -0.11274    0.0051774
    {'[1254.28,1597.44)'}      76      44    1.7273    -0.15787    0.0025554
    {'[1597.44,Inf]'     }     72      48       1.5    -0.29895    0.0093402
    {'Totals'            }    803     397    2.0227         NaN     0.039868
```

```
cp = 3×1
10³ ×

    0.5589
    1.2543
    1.5974
```

```
cp(2) = []; % To merge bins 2 and 3
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-08
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:
    logit(status) ~ 1 + CustAge + ResStatus + EmpStatus + CustIncome + TmWBank + OtherCC + AMBala
    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
    {'CustAge'   }    {'<missing>'       }          NaN
    {'ResStatus' }    {'Tenant'          }    -0.031302
    {'ResStatus' }    {'Home Owner'      }      0.12697
    {'ResStatus' }    {'Other'           }      0.37652
    {'ResStatus' }    {'<missing>'       }          NaN
    {'EmpStatus' }    {'Unknown'         }    -0.076369
    {'EmpStatus' }    {'Employed'        }      0.31456
    {'EmpStatus' }    {'<missing>'       }          NaN
    {'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
    {'CustIncome'}    {'<missing>'       }          NaN
    {'TmWBank'   }    {'[-Inf,12)'       }     -0.18221
    {'TmWBank'   }    {'[12,45)'          }    -0.038279
    {'TmWBank'   }    {'[45,71)'          }      0.39569
    {'TmWBank'   }    {'[71,Inf]'         }      0.95074
    {'TmWBank'   }    {'<missing>'        }          NaN
    {'OtherCC'   }    {'No'               }       -0.193
    {'OtherCC'   }    {'Yes'              }      0.15868
    {'OtherCC'   }    {'<missing>'        }          NaN
    {'AMBalance' }    {'[-Inf,558.88)'    }       0.3552
    {'AMBalance' }    {'[558.88,1597.44)'}    -0.026797
    {'AMBalance' }    {'[1597.44,Inf]'    }     -0.21168
    {'AMBalance' }    {'<missing>'        }          NaN
```

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
    {'CustAge'   }    {'<missing>'        }          NaN
    {'ResStatus' }    {'Tenant'           }    -0.031302
    {'ResStatus' }    {'Home Owner'       }      0.12697
    {'ResStatus' }    {'Other'            }      0.37652
    {'ResStatus' }    {'<missing>'        }          NaN
    {'EmpStatus' }    {'Unknown'          }    -0.076369
    {'EmpStatus' }    {'Employed'         }      0.31456
    {'EmpStatus' }    {'<missing>'        }          NaN
    {'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
    {'CustIncome'}    {'<missing>'        }          NaN
    {'TmWBank'   }    {'Up to 11'         }     -0.18221
    {'TmWBank'   }    {'12 to 44'         }    -0.038279
    {'TmWBank'   }    {'45 to 70'         }      0.39569
    {'TmWBank'   }    {'71 and up'        }      0.95074
    {'TmWBank'   }    {'<missing>'        }          NaN
    {'OtherCC'   }    {'No'               }       -0.193
    {'OtherCC'   }    {'Yes'                }      0.15868
    {'OtherCC'   }    {'<missing>'          }          NaN
    {'AMBalance' }    {'Up to 558.87'       }       0.3552
    {'AMBalance' }    {'558.88 to 1597.43'}    -0.026797
    {'AMBalance' }    {'1597.44 and up'     }     -0.21168
    {'AMBalance' }    {'<missing>'          }          NaN
```

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
    {'CustAge'   }    {'<missing>'        }       NaN
    {'ResStatus' }    {'Tenant'           }    62.028
    {'ResStatus' }    {'Home Owner'       }    73.445
    {'ResStatus' }    {'Other'            }    91.446
    {'ResStatus' }    {'<missing>'        }       NaN
    {'EmpStatus' }    {'Unknown'          }    58.777
    {'EmpStatus' }    {'Employed'         }    86.976
    {'EmpStatus' }    {'<missing>'        }       NaN
    {'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
    {'CustIncome'}    {'<missing>'        }       NaN
    {'TmWBank'   }    {'Up to 11'         }    51.142
    {'TmWBank'   }    {'12 to 44'         }    61.524
    {'TmWBank'   }    {'45 to 70'         }    92.829
    {'TmWBank'   }    {'71 and up'        }    132.87
    {'TmWBank'   }    {'<missing>'        }       NaN
    {'OtherCC'   }    {'No'               }    50.364
    {'OtherCC'   }    {'Yes'              }    75.732
    {'OtherCC'   }    {'<missing>'        }       NaN
    {'AMBalance' }    {'Up to 558.87'     }    89.908
    {'AMBalance' }    {'558.88 to 1597.43'}    62.353
    {'AMBalance' }    {'1597.44 and up'   }    49.016
    {'AMBalance' }    {'<missing>'        }       NaN
```

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))
```

```
  528.2044
  554.8861
  505.2406
  564.0717
  554.8861
  586.1904
  441.8755
  515.8125
  524.4553
  508.3169
```

```
disp(Points(1:10,:))
```

```
    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.

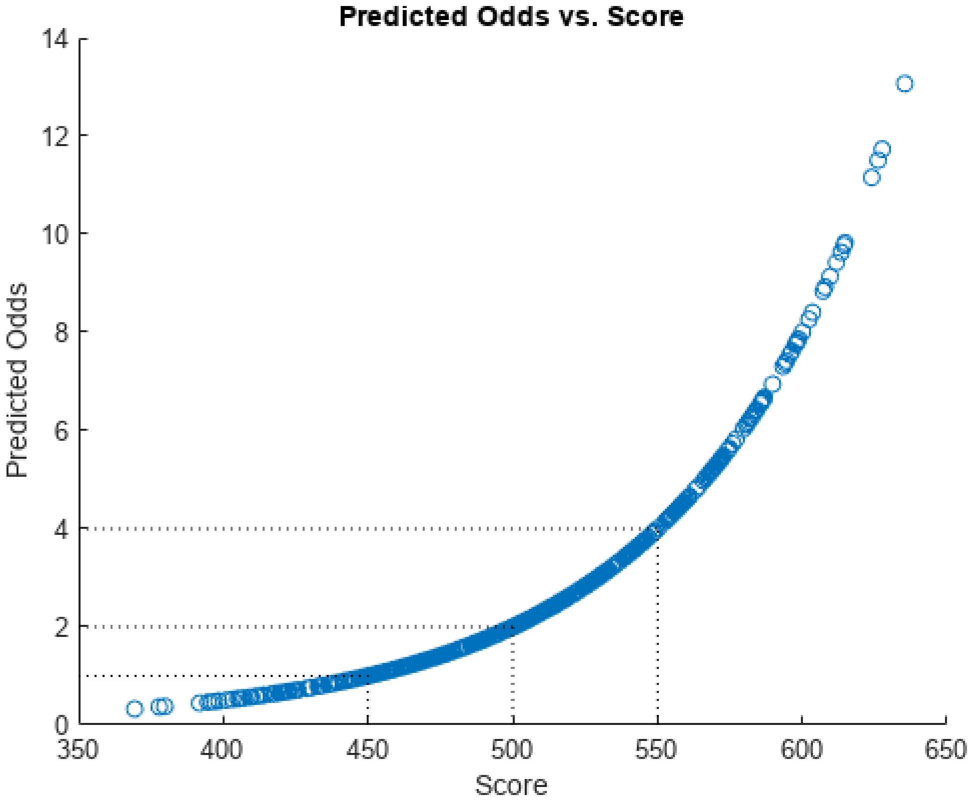
To calculate the probability of default, use the `probdefault` function.

```
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;
```

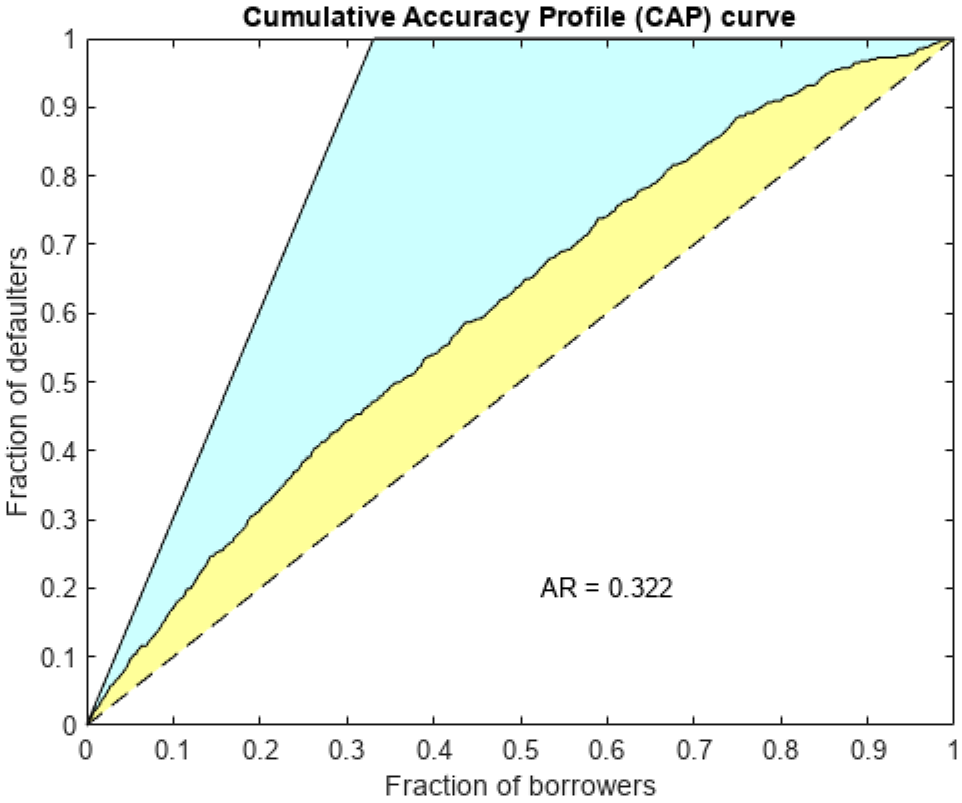
```
yLimits = ylim;

% 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
```

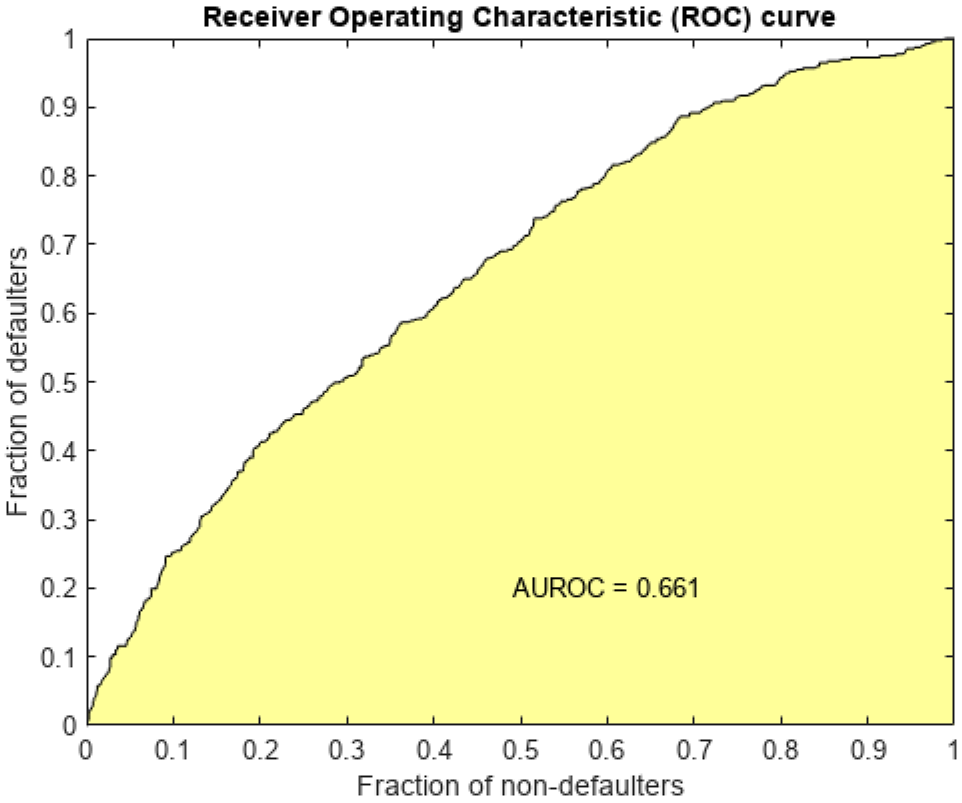


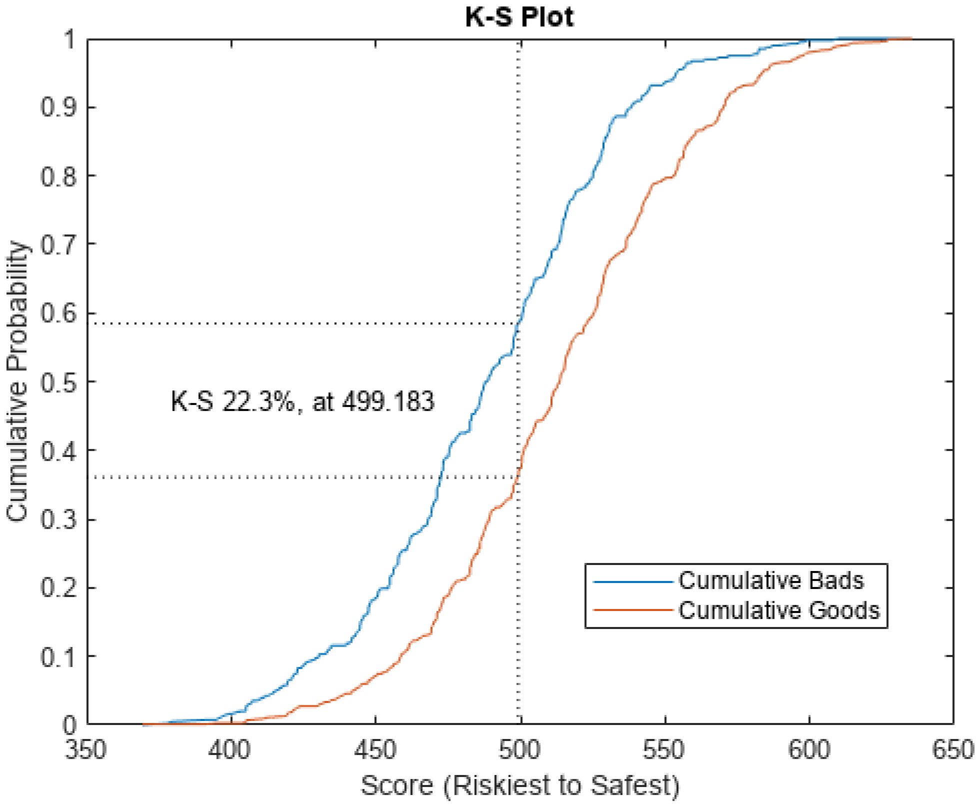
Step 7. Validate the credit scorecard model using the CAP, ROC, and Kolmogorov-Smirnov statistic

The `creditscorecard` class supports three validation methods, the Cumulative Accuracy Profile (CAP), the Receiver Operating Characteristic (ROC), and the Kolmogorov-Smirnov (KS) statistic. For more information on CAP, ROC, and KS, see “Cumulative Accuracy Profile (CAP)” on page 15-1867, “Receiver Operating Characteristic (ROC)” on page 15-1867, and “Kolmogorov-Smirnov statistic (KS)” on page 15-1867.

```
[Stats,T] = validatemodel(sc,'Plot',{'CAP','ROC','KS'});
```


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



```
disp(Stats)
```

```
            Measure                 Value 
    ________________________     _______

    {'Accuracy Ratio'       }    0.32225
    {'Area under ROC curve'}    0.66113
    {'KS statistic'         }    0.22324
    {'KS score'             }     499.18
```

```
disp(T(1:15,:))
```

```
    Scores    ProbDefault    TrueBads    FalseBads    TrueGoods    FalseGoods    Sensitivity
    ______    ___________    ________    _________    _________    __________    ___________

     369.4       0.7535          0           1           802           397                 0
    377.86      0.73107          1           1           802           396         0.0025189
    379.78       0.7258          2           1           802           395         0.0050378
    391.81      0.69139          3           1           802           394         0.0075567
    394.77      0.68259          3           2           801           394         0.0075567
    395.78      0.67954          4           2           801           393          0.010076
    396.95      0.67598          5           2           801           392          0.012594
    398.37      0.67167          6           2           801           391          0.015113
    401.26      0.66276          7           2           801           390          0.017632
    403.23      0.65664          8           2           801           389          0.020151
    405.09      0.65081          8           3           800           389          0.020151
    405.15      0.65062         11           5           798           386          0.027708
    405.37      0.64991         11           6           797           386          0.027708
    406.18      0.64735          12           6           797           385          0.030227
    407.14      0.64433          13           6           797           384          0.032746
```

See Also

`creditscorecard` | `autobinning` | `bininfo` | `predictorinfo` | `modifypredictor` | `modifybins` | `bindata` | `plotbins` | `fitmodel` | `displaypoints` | `formatpoints` | `score` | `setmodel` | `probdefault` | `validatemodel` | `compact`

Related Examples

- “Troubleshooting Credit Scorecard Results” on page 8-63
- “Credit Rating by Bagging Decision Trees”

More About

- “About Credit Scorecards” on page 8-47
- “Credit Scorecard Modeling Workflow” on page 8-51
- “Credit Scorecard Modeling Using Observation Weights” on page 8-54
- Monotone Adjacent Pooling Algorithm (MAPA) on page 15-1828

External Websites

- Credit Risk Modeling with MATLAB (53 min 10 sec)

Credit Scorecards with Constrained Logistic Regression Coefficients

To compute scores for a creditscorecard object with constraints for equality, inequality, or bounds on the coefficients of the logistic regression model, use fitConstrainedModel. Unlike fitmodel, fitConstrainedModel solves for both the unconstrained and constrained problem. The current solver used to minimize an objective function for fitConstrainedModel is fmincon, from the Optimization Toolbox™.

This example has three main sections. First, fitConstrainedModel is used to solve for the coefficients in the unconstrained model. Then, fitConstrainedModel demonstrates how to use several types of constraints. Finally, fitConstrainedModel uses bootstrapping for the significance analysis to determine which predictors to reject from the model.

Create the creditscorecard Object and Bin data

```
load CreditCardData.mat
sc = creditscorecard(data,'IDVar','CustID');
sc = autobinning(sc);
```

Unconstrained Model Using fitConstrainedModel

Solve for the unconstrained coefficients using fitConstrainedModel with default values for the input parameters. fitConstrainedModel uses the internal optimization solver fmincon from the Optimization Toolbox™. If you do not set any constraints, fmincon treats the model as an unconstrained optimization problem. The default parameters for the LowerBound and UpperBound are -Inf and +Inf, respectively. For the equality and inequality constraints, the default is an empty numeric array.

```
[sc1,mdl1] = fitConstrainedModel(sc);
coeff1 = mdl1.Coefficients.Estimate;
disp(mdl1.Coefficients);
```

```
                   Estimate 
                   _________

    (Intercept)      0.70246
    CustAge           0.6057
    TmAtAddress       1.0381
    ResStatus         1.3794
    EmpStatus        0.89648
    CustIncome       0.70179
    TmWBank           1.1132
    OtherCC           1.0598
    AMBalance         1.0572
    UtilRate       -0.047597
```

Unlike fitmodel which gives *p*-values, when using fitConstrainedModel, you must use bootstrapping to find out which predictors are rejected from the model, when subject to constraints. This is illustrated in the "Significance Bootstrapping" section.

Using fitmodel to Compare the Results and Calibrate the Model

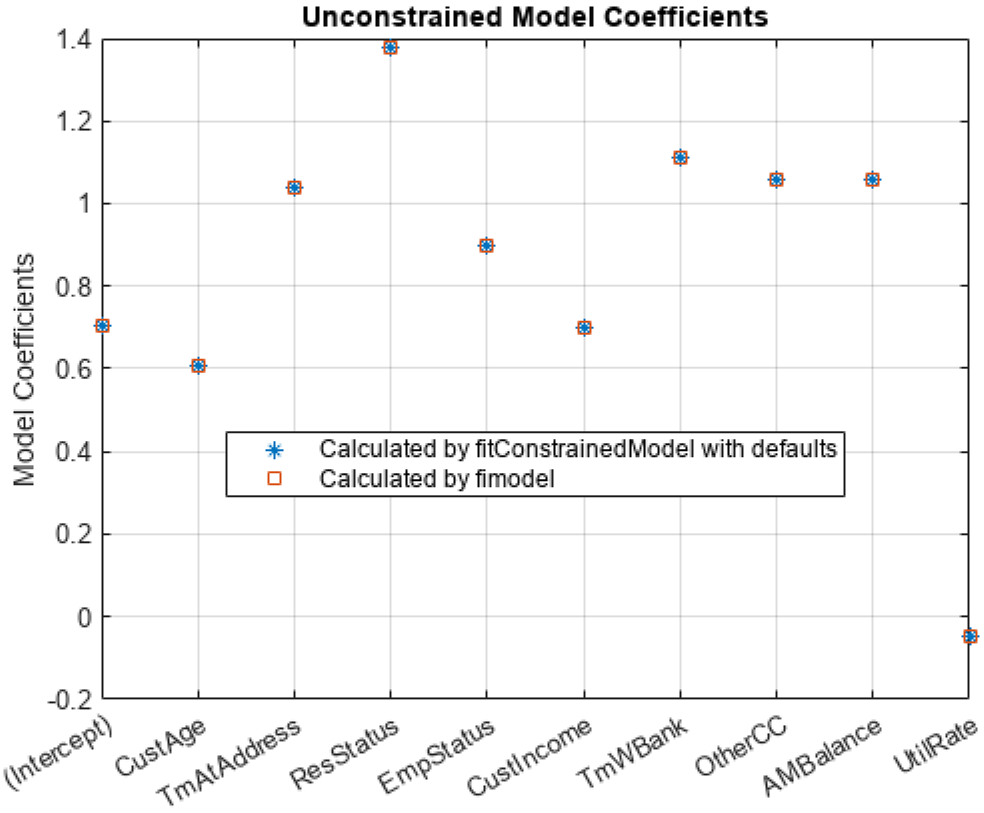
fitmodel fits a logistic regression model to the Weight-of-Evidence (WOE) data and there are no constraints. You can compare the results from the "Unconstrained Model Using fitConstrainedModel" section with those of fitmodel to verify that the model is well calibrated.

Now, solve the unconstrained problem by using fitmodel. Note that fitmodel and fitConstrainedModel use different solvers. While fitConstrainedModel uses fmincon, fitmodel uses stepwiseglm by default. To include all predictors from the start, set the 'VariableSelection' name-value pair argument of fitmodel to 'fullmodel'.

```
[sc2,mdl2] = fitmodel(sc,'VariableSelection','fullmodel','Display','off');
coeff2 = mdl2.Coefficients.Estimate;
disp(mdl2.Coefficients);
```

```
                   Estimate       SE         tStat       pValue   
                   _________    ________    _________    __________

    (Intercept)      0.70246    0.064039       10.969    5.3719e-28
    CustAge           0.6057     0.24934       2.4292      0.015131
    TmAtAddress       1.0381     0.94042       1.1039       0.26963
    ResStatus         1.3794      0.6526       2.1137      0.034538
    EmpStatus        0.89648     0.29339       3.0556     0.0022458
    CustIncome       0.70179     0.21866       3.2095     0.0013295
    TmWBank           1.1132     0.23346       4.7683    1.8579e-06
    OtherCC           1.0598     0.53005       1.9994      0.045568
    AMBalance         1.0572     0.36601       2.8884     0.0038718
    UtilRate       -0.047597     0.61133    -0.077858       0.93794
```

```
figure
plot(coeff1,'*')
hold on
plot(coeff2,'s')
xticklabels(mdl1.Coefficients.Properties.RowNames)
ylabel('Model Coefficients')
title('Unconstrained Model Coefficients')
legend({'Calculated by fitConstrainedModel with defaults','Calculated by fimodel'},'Location','be
grid on
```



As both the tables and the plot show, the model coefficients match. You can be confident that this implementation of `fitConstrainedModel` is well calibrated.

Constrained Model

In the constrained model approach, you solve for the values of the coefficients b_i of the logistic model, subject to constraints. The supported constraints are bound, equality, or inequality. The coefficients maximize the likelihood-of-default function defined, for observation i, as:

$$L_i = p(\text{Default}_i)^{y_i} \times (1 - p(\text{Default}_i))^{1 - y_i}$$

where:

- $$p(\text{Default}_i) = \frac{1}{1 + e^{-b\text{x}_i}}$$
- $\boldsymbol{b} = [b_1\, b_2\, . . .\, b_K]$ is an unknown model coefficient
- $\mathbf{x}_{\boldsymbol{i}} = [x_{\text{i}1}\, x_2\, . . .\, x_{\text{iK}}]$ is the predictor values at observation i
- y_i is the response value; a value of 1 represents default and a value of 0 represents non-default

This formula is for non-weighted data. When observation i has weight w_i, it means that there are w_i as many observations i. Therefore, the probability that default occurs at observation i is the product of the probabilities of default:

$$p_i \; = \; \underbrace{p(\text{Default}_i)^{y_i} * p(\text{Default}_i)^{y_i} * \; . \; . \; . \; * p(\text{Default}_i)^{y_i}}_{w_i \, \text{times}} = p(\text{Default}_i)^{w_i * y_i}$$

Likewise, the probability of non-default for weighted observation i is:

$$\widehat{p}_i \; = \; \underbrace{p(\sim\text{Default}_i)^{1 - y_i} * p(\sim\text{Default}_i)^{1 - y_i} * \; . \; . \; . \; * p(\sim\text{Default}_i)^{1 - y_i}}_{w_i \, \text{times}} = (1 - p(\text{Default}_i))^{w_i * (1 - y_i)}$$

For weighted data, if there is default at a given observation i whose weight is w_i, it is as if there was a w_i count of that one observation, and all of them either all default, or all non-default. w_i may or may not be an integer.

Therefore, for the weighted data, the likelihood-of-default function for observation i in the first equation becomes

$$L_i = p(\text{Default}_i)^{w_i * y_i} \times (1 - p(\text{Default}_i))^{w_i * (1 - y_i)}$$

By assumption, all defaults are independent events, so the objective function is

$$L = L_1 \times L_2 \times \; . \; . \; . \; \times \; L_N$$

or, in more convenient logarithmic terms:

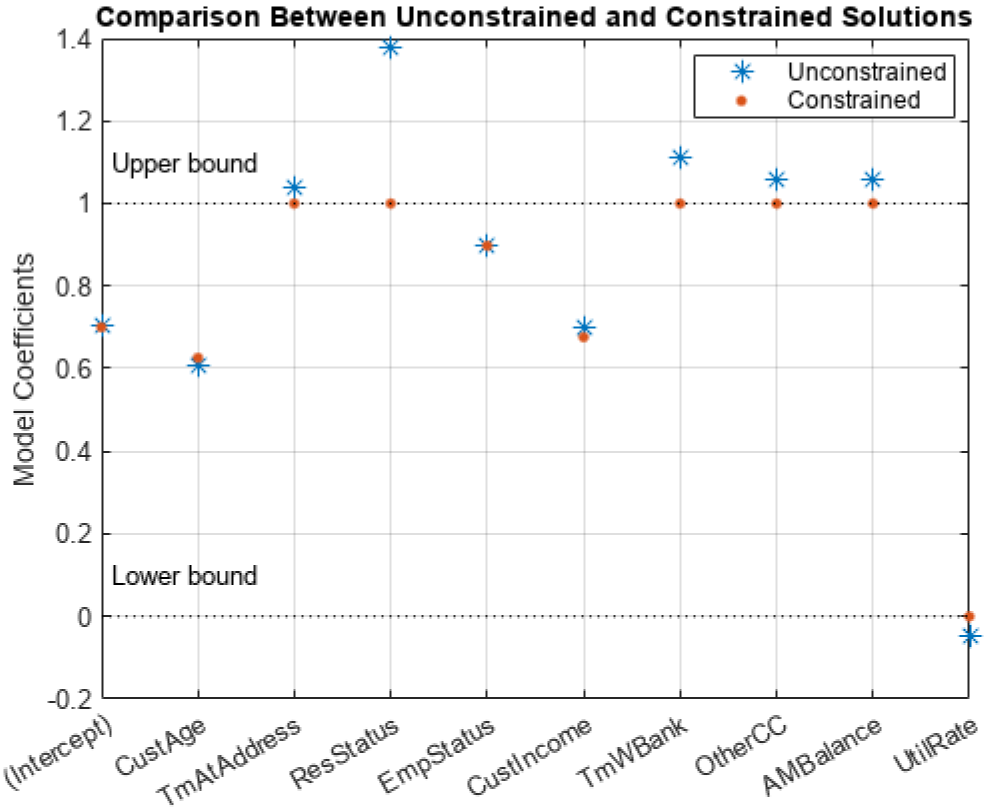
$$\log(L) = \sum_{i = 1}^{N} w_i * [y_i \log(p(\text{Default}_i)) + (1 - y_i)\log(1 - p(\text{Default}_i))]$$

Apply Constraints on the Coefficients

After calibrating the unconstrained model as described in the "Unconstrained Model Using fitConstrainedModel" section, you can solve for the model coefficients subject to constraints. You can choose lower and upper bounds such that $0 \leq b_i \leq 1, \forall i = 1 \; . \; . \; . \; K$, except for the intercept. Also, since the customer age and customer income are somewhat correlated, you can also use additional constraints on their coefficients, for example, $|b_{CusAge} - b_{CustIncome}| < 0.1$. The coefficients corresponding to the predictors `'CustAge'` and `'CustIncome'` in this example are b_2 and b_6, respectively.

```
K  = length(sc.PredictorVars);
lb = [-Inf;zeros(K,1)];
ub = [Inf;ones(K,1)];
AIneq = [0 -1 0 0 0 1 0 0 0 0;0 -1 0 0 0 -1 0 0 0 0];
bIneq = [0.05;0.05];
Options = optimoptions('fmincon','SpecifyObjectiveGradient',true,'Display','off');
[sc3,mdl] = fitConstrainedModel(sc,'AInequality',AIneq,'bInequality',bIneq,...
    'LowerBound',lb,'UpperBound',ub,'Options',Options);

figure
plot(coeff1,'*','MarkerSize',8)
hold on
plot(mdl.Coefficients.Estimate,'.','MarkerSize',12)
line(xlim,[0 0],'color','k','linestyle',':')
line(xlim,[1 1],'color','k','linestyle',':')
text(1.1,0.1,'Lower bound')
text(1.1,1.1,'Upper bound')
grid on
```

```
xticklabels(mdl.Coefficients.Properties.RowNames)
ylabel('Model Coefficients')
title('Comparison Between Unconstrained and Constrained Solutions')
legend({'Unconstrained','Constrained'},'Location','best')
```



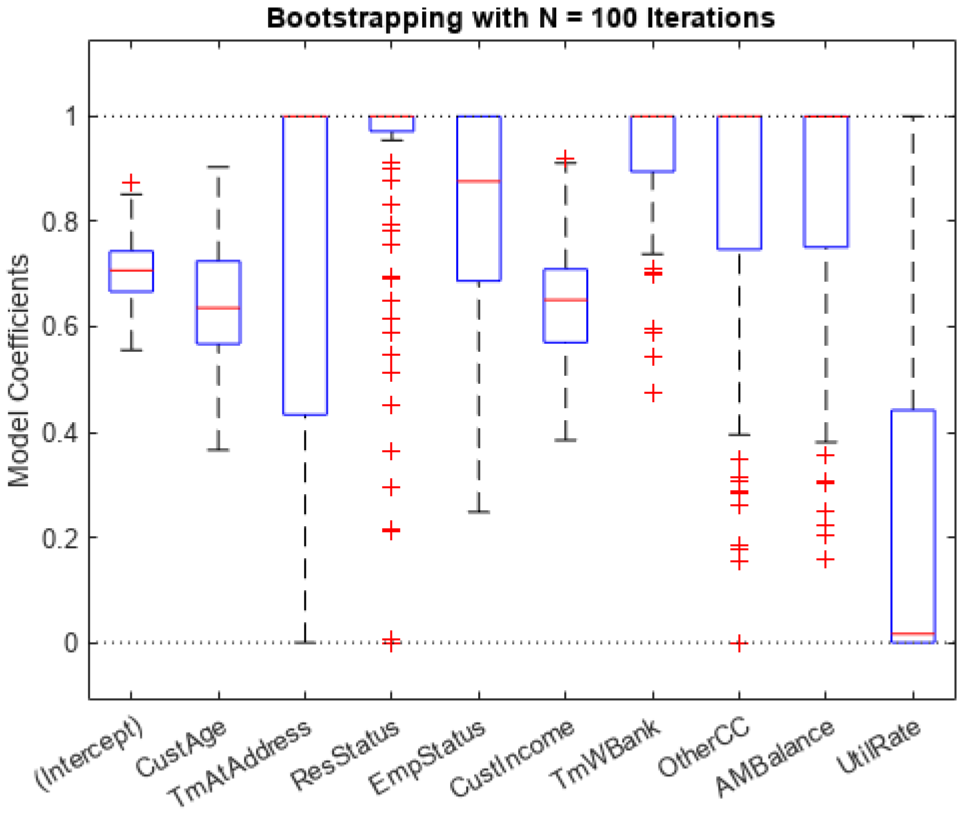
Significance Bootstrapping

For the unconstrained problem, standard formulas are available for computing *p*-values, which you use to evaluate which coefficients are significant and which are to be rejected. However, for the constrained problem, standard formulas are not available, and the derivation of formulas for significance analysis is complicated. A practical alternative is to perform significance analysis through *bootstrapping*.

In the bootstrapping approach, when using `fitConstrainedModel`, you set the name-value argument `'Bootstrap'` to `true` and chose a value for the name-value argument `'BootstrapIter'`. Bootstrapping means that *NIter* samples (with replacement) from the original observations are selected. In each iteration, `fitConstrainedModel` solves for the same constrained problem as the "Constrained Model" section. `fitConstrainedModel` obtains several values (solutions) for each coefficient b_i and you can plot these as a `boxplot` or `histogram`. Using the boxplot or histogram, you can examine the median values to evaluate whether the coefficients are away from zero and how much the coefficients deviate from their means.

```
lb = [-Inf;zeros(K,1)];
ub = [Inf;ones(K,1)];
AIneq = [0 -1 0 0 0 1 0 0 0 0;0 1 0 0 0 -1 0 0 0 0];
bIneq = [0.05;0.05];
```

```
c0 = zeros(K,1);
NIter = 100;
Options = optimoptions('fmincon','SpecifyObjectiveGradient',true,'Display','off');
rng('default')

[sc,mdl] = fitConstrainedModel(sc,'AInequality',AIneq,'bInequality',bIneq,...
    'LowerBound',lb,'UpperBound',ub,'Bootstrap',true,'BootstrapIter',NIter,'Options',Options);

figure
boxplot(mdl.Bootstrap.Matrix,mdl.Coefficients.Properties.RowNames)
hold on
line(xlim,[0 0],'color','k','linestyle',':')
line(xlim,[1 1],'color','k','linestyle',':')
title('Bootstrapping with N = 100 Iterations')
ylabel('Model Coefficients')
```

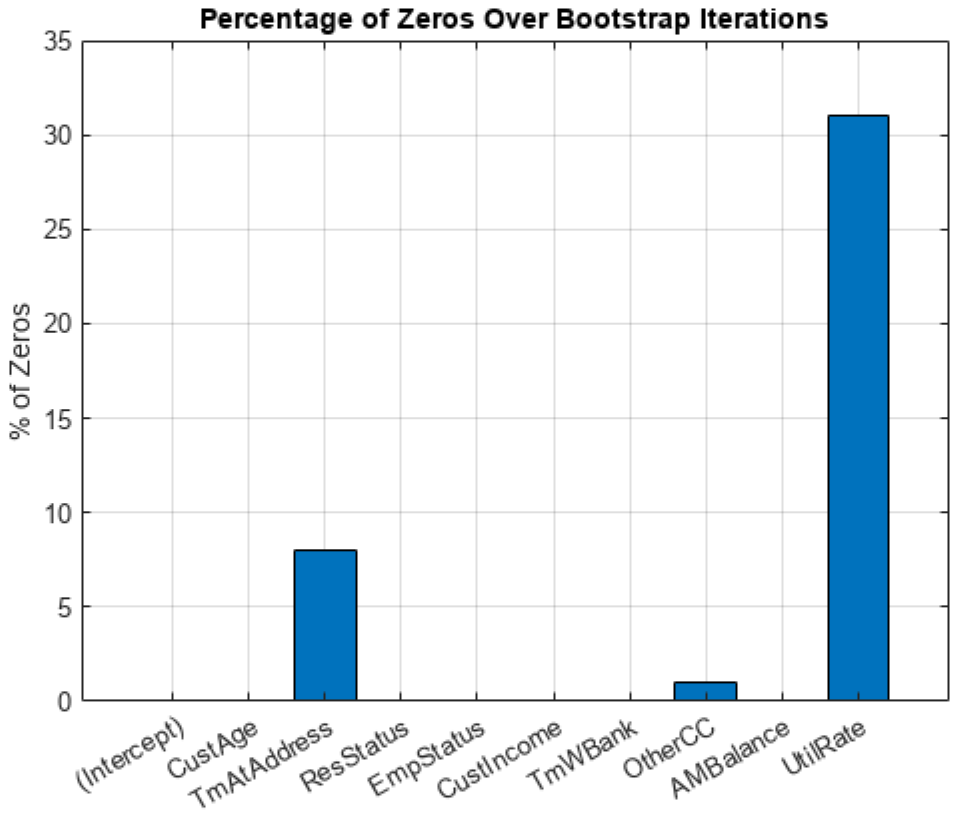


The solid red lines in the boxplot indicate that the median values and the bottom and top edges are for the 25th and 75th percentiles. The "whiskers" are the minimum and maximum values, not including outliers. The dotted lines represent the lower and upper bound constraints on the coefficients. In this example, the coefficients cannot be negative, by construction.

To help decide which predictors to keep in the model, assess the proportion of times each coefficient is zero.

```
Tol = 1e-6;
figure
bar(100*sum(mdl.Bootstrap.Matrix<= Tol)/NIter)
```

```
ylabel('% of Zeros')
title('Percentage of Zeros Over Bootstrap Iterations')
xticklabels(mdl.Coefficients.Properties.RowNames)
grid on
```

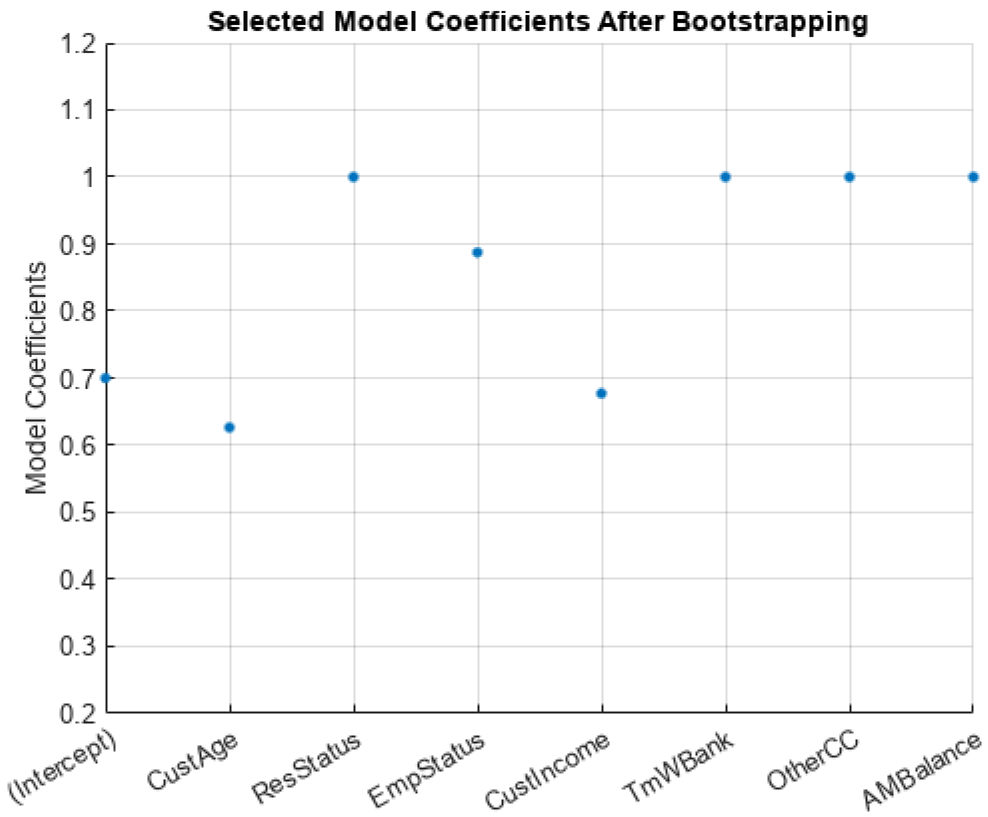


Based on the plot, you can reject `'UtilRate'` since it has the highest number of zero values. You can also decide to reject `'TmAtAddress'` since it shows a peak, albeit small.

Set the Corresponding Coefficients to Zero

To set the corresponding coefficients to zero, set their upper bound to zero and solve the model again using the original data set.

```
ub(3) = 0;
ub(end) = 0;
[sc,mdl] = fitConstrainedModel(sc,'AInequality',AIneq,'bInequality',bIneq,'LowerBound',lb,'Upper
Ind = (abs(mdl.Coefficients.Estimate) <= Tol);
ModelCoeff = mdl.Coefficients.Estimate(~Ind);
ModelPreds = mdl.Coefficients.Properties.RowNames(~Ind)';

figure
hold on
plot(ModelCoeff,'.','MarkerSize',12)
ylim([0.2 1.2])
ylabel('Model Coefficients')
xticklabels(ModelPreds)
title('Selected Model Coefficients After Bootstrapping')
grid on
```



Set Constrained Coefficients Back Into the `creditscorecard`

Now that you have solved for the constrained coefficients, use `setmodel` to set the model's coefficients and predictors. Then you can compute the (unscaled) points.

```
ModelPreds = ModelPreds(2:end);
sc = setmodel(sc,ModelPreds,ModelCoeff);
p = displaypoints(sc);

disp(p)
```

```
      Predictors              Bin              Points
    _______________    ______________________    _________

    {'CustAge'    }    {'[-Inf,33)'       }     -0.16725
    {'CustAge'    }    {'[33,37)'         }     -0.14811
    {'CustAge'    }    {'[37,40)'         }    -0.065607
    {'CustAge'    }    {'[40,46)'         }     0.044404
    {'CustAge'    }    {'[46,48)'         }      0.21761
    {'CustAge'    }    {'[48,58)'         }      0.23404
    {'CustAge'    }    {'[58,Inf]'        }      0.49029
    {'CustAge'    }    {'<missing>'       }          NaN
    {'ResStatus' }    {'Tenant'          }    0.0044307
    {'ResStatus' }    {'Home Owner'      }      0.11932
    {'ResStatus' }    {'Other'           }      0.30048
    {'ResStatus' }    {'<missing>'       }          NaN
    {'EmpStatus' }    {'Unknown'         }    -0.077028
    {'EmpStatus' }    {'Employed'        }      0.31459
    {'EmpStatus' }    {'<missing>'        }          NaN
    {'CustIncome'}    {'[-Inf,29000)'     }     -0.43795
    {'CustIncome'}    {'[29000,33000)'    }    -0.097814
    {'CustIncome'}    {'[33000,35000)'    }     0.053667
    {'CustIncome'}    {'[35000,40000)'    }     0.081921
    {'CustIncome'}    {'[40000,42000)'    }     0.092364
    {'CustIncome'}    {'[42000,47000)'    }      0.23932
    {'CustIncome'}    {'[47000,Inf]'      }      0.42477
    {'CustIncome'}    {'<missing>'        }          NaN
    {'TmWBank'   }    {'[-Inf,12)'        }     -0.15547
    {'TmWBank'   }    {'[12,23)'          }    -0.031077
    {'TmWBank'   }    {'[23,45)'          }    -0.021091
    {'TmWBank'   }    {'[45,71)'          }      0.36703
    {'TmWBank'   }    {'[71,Inf]'         }      0.86888
    {'TmWBank'   }    {'<missing>'        }          NaN
    {'OtherCC'   }    {'No'               }     -0.16832
    {'OtherCC'   }    {'Yes'              }      0.15336
    {'OtherCC'   }    {'<missing>'        }          NaN
    {'AMBalance' }    {'[-Inf,558.88)'    }      0.34418
    {'AMBalance' }    {'[558.88,1254.28)' }    -0.012745
    {'AMBalance' }    {'[1254.28,1597.44)'}    -0.057879
    {'AMBalance' }    {'[1597.44,Inf]'    }     -0.19896
    {'AMBalance' }    {'<missing>'        }          NaN
```

Using the unscaled points, you can follow the remainder of the “Credit Scorecard Modeling Workflow” on page 8-51 to compute scores and probabilities of default and to validate the model.

See Also

`creditscorecard` | `autobinning` | `bininfo` | `predictorinfo` | `modifypredictor` | `modifybins` | `bindata` | `plotbins` | `fitmodel` | `displaypoints` | `formatpoints` | `score` | `setmodel` | `probdefault` | `validatemodel` | `compact`

Related Examples

- “Troubleshooting Credit Scorecard Results” on page 8-63
- “Credit Rating by Bagging Decision Trees”

More About

- “About Credit Scorecards” on page 8-47
- “Credit Scorecard Modeling Workflow” on page 8-51
- “Credit Scorecard Modeling Using Observation Weights” on page 8-54
- Monotone Adjacent Pooling Algorithm (MAPA) on page 15-1828

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 the following objects and functions:

CDS Objects

Object	Purpose
`CDS`	`CDS` instrument object.
`CDSOption`	`CDSOption` instrument object.

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` | `cdsspread` | `cdsrpv01`

Related Examples

- "First-to-Default Swaps" (Financial Instruments Toolbox)
- "Credit Default Swap Option" (Financial Instruments Toolbox)
- "Counterparty Credit Risk and CVA" (Financial Instruments Toolbox)
- "Bootstrapping a Default Probability Curve from Credit Default Swaps" (Financial Instruments Toolbox)

More About

- "Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments" (Financial Instruments Toolbox)

Bootstrapping a Default Probability Curve

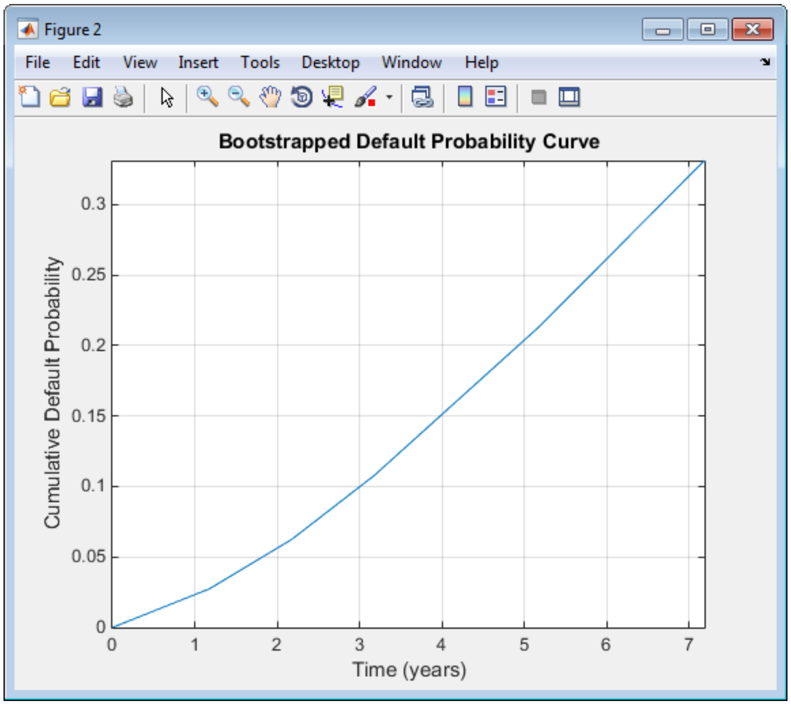
This example shows how to bootstrap default probabilities from CDS market quotes. To bootstrap default probabilities from bond market data, see `bondDefaultBootstrap`. In a typical workflow, pricing a new CDS contract involves first estimating a default probability term structure using `cdsbootstrap`. 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);
```

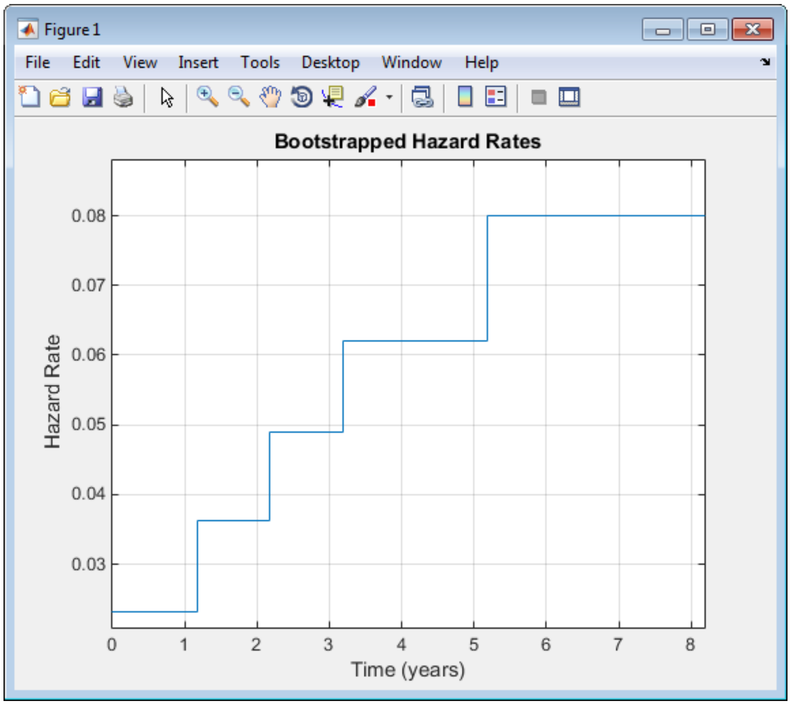
The bootstrapped default probability curve is plotted against time, in years, from the valuation date.

```
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, and so on, 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

`cdsbootstrap` | `cdsprice` | `cdsspread` | `cdsrpv01` | `bondDefaultBootstrap`

Related Examples

- “First-to-Default Swaps” (Financial Instruments Toolbox)
- “Credit Default Swap Option” (Financial Instruments Toolbox)
- “Counterparty Credit Risk and CVA” (Financial Instruments Toolbox)

Finding Breakeven Spread for New CDS Contract

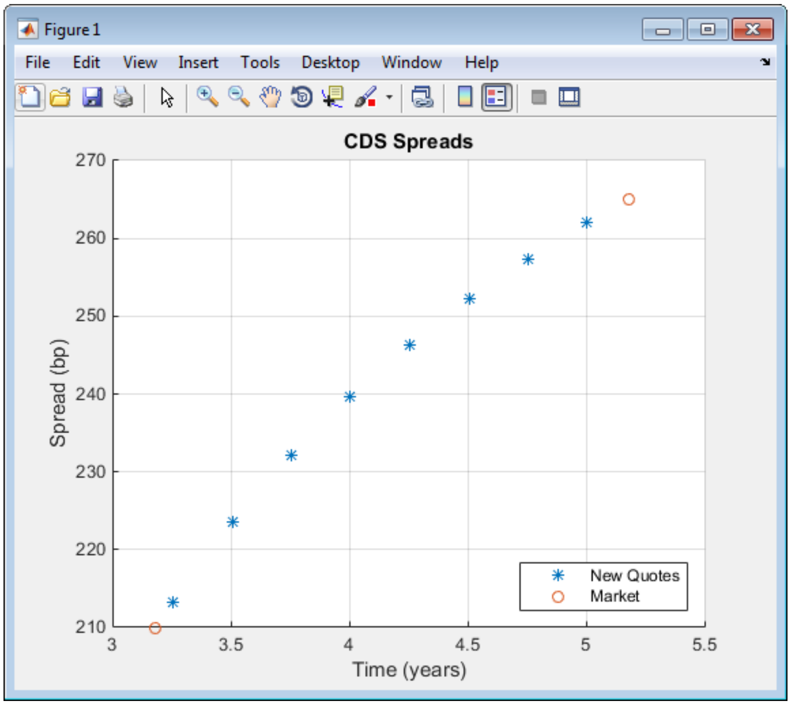
The breakeven, or running, spread is the premium a protection buyer must pay, with no upfront payments involved, to receive protection for credit events associated to a given reference entity. Spreads are expressed in basis points (bp). There is a notional amount associated to the CDS contract to determine 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 three to five 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')
```

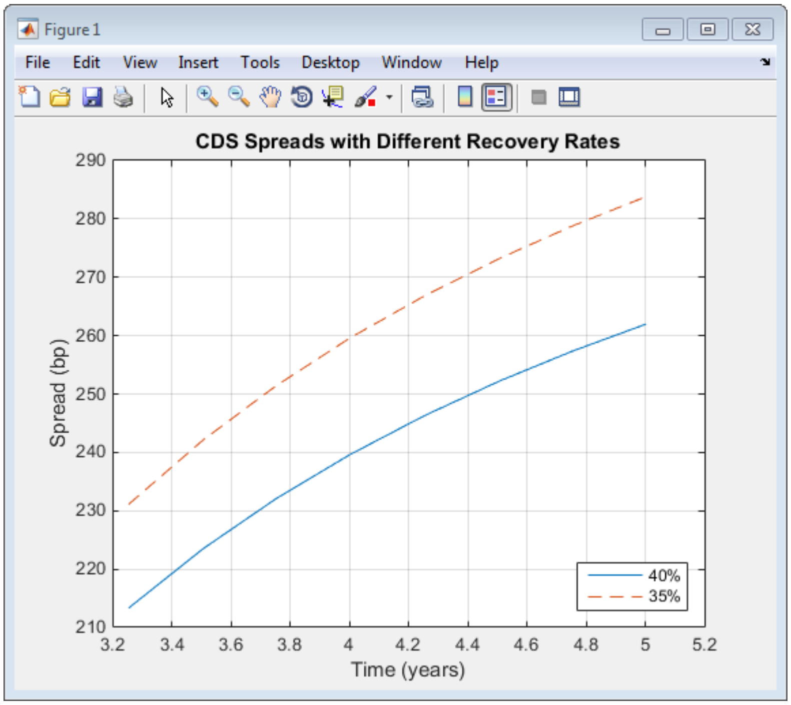
This plot displays the resulting spreads:



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` | `cdsspread` | `cdsrpv01`

Related Examples

- “First-to-Default Swaps” (Financial Instruments Toolbox)
- “Credit Default Swap Option” (Financial Instruments Toolbox)
- “Counterparty Credit Risk and CVA” (Financial Instruments Toolbox)

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+AccPrem);
fprintf('Accrued Premium: %8.2f\n',AccPrem);
fprintf('Clean Price: %8.2f\n',Price);
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: 56872.94
Accrued Premium: 15244.44
Clean Price: 41628.50

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
```

Also, 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), AccPrem2(1), Price2(1));
fprintf('   Short \t $ %8.2f \t $ %8.2f \t $ %8.2f \t\n',...
   Price2(2)+AccPrem2(2), AccPrem2(2), Price2(2));
fprintf('Mark-to-market of hedged position: $ %8.2f\n',sum(Price2)+sum(AccPrem2));
```

This resulting value of the portfolio is:

```
Contract      Dirty Price      Acc Premium       Clean Price
    Long      $  56872.94      $  15244.44      $  41628.50
   Short      $ -48185.88      $ -15477.78      $ -32708.11
Mark-to-market of hedged position: $  8687.06
```

See Also

`cdsbootstrap` | `cdsprice` | `cdsspread` | `cdsrpv01`

Related Examples

- “First-to-Default Swaps” (Financial Instruments Toolbox)
- “Credit Default Swap Option” (Financial Instruments Toolbox)
- “Counterparty Credit Risk and CVA” (Financial Instruments Toolbox)

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 four-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-5), the FHR approach is a quoting convention only, and leads to quotes inconsistent with market data. For example, calculating the upfront for the three-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 | cdsspread | cdsrpv01

Related Examples

- “First-to-Default Swaps” (Financial Instruments Toolbox)
- “Credit Default Swap Option” (Financial Instruments Toolbox)
- “Counterparty Credit Risk and CVA” (Financial Instruments Toolbox)

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)
```

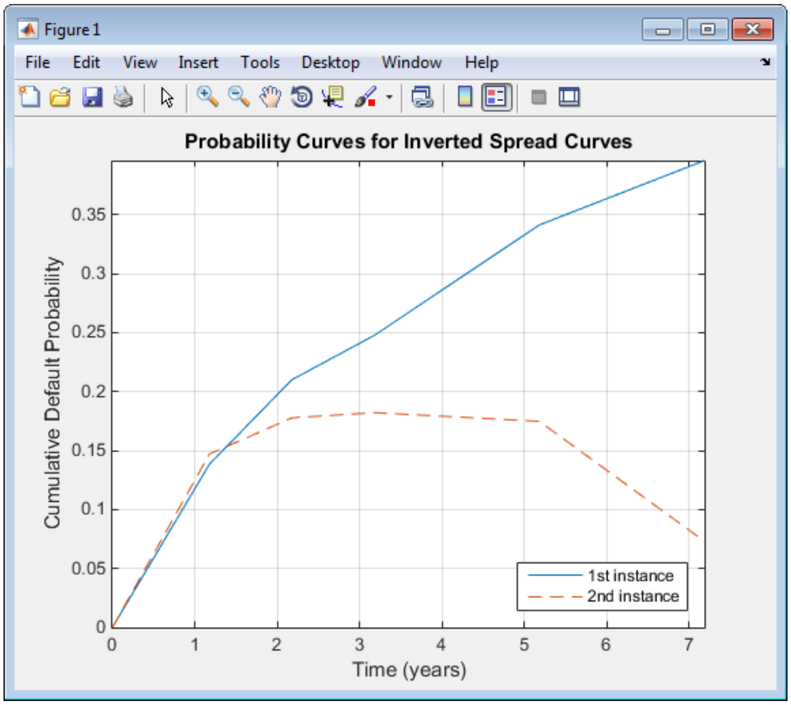
A non-monotone bootstrapped probability curve implies negative default probabilities and negative hazard rates for certain time intervals. Extreme market conditions can lead to these types of situations. In these cases, you must assess the reliability and usefulness of the bootstrapped results.

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')
```

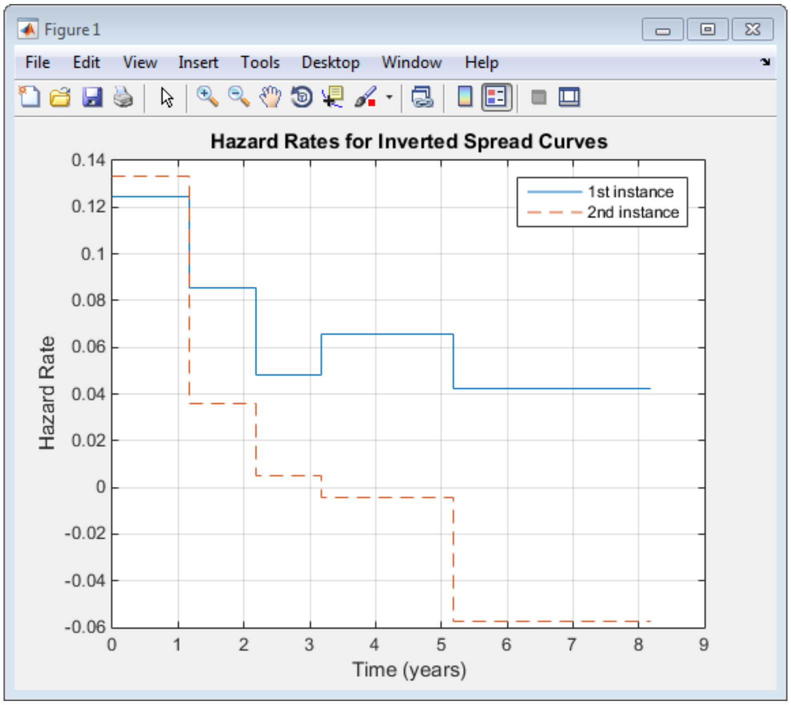
The resulting plot



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-5).

See Also

`cdsbootstrap` | `cdsprice` | `cdsspread` | `cdsrpv01`

Related Examples

- "First-to-Default Swaps" (Financial Instruments Toolbox)
- "Credit Default Swap Option" (Financial Instruments Toolbox)
- "Counterparty Credit Risk and CVA" (Financial Instruments Toolbox)

Visualize Transitions Data for transprob

This example shows how to visualize credit rating transitions that are used as an input to the `transprob` function. The example also describes how the `transprob` function treats rating transitions when the company data starts after the start date of the analysis, or when the end date of the analysis is after the last transition observed.

Sample Data

Set up fictitious sample data for illustration purposes.

```
data = {'ABC','17-Feb-2015','AA';
    'ABC','6-Jul-2017','A';
    'LMN','12-Aug-2014','B';
    'LMN','9-Nov-2015','CCC';
    'LMN','7-Sep-2016','D';
    'XYZ','14-May-2013','BB';
    'XYZ','21-Jun-2016','BBB'};
data = cell2table(data,'VariableNames',{'ID','Date','Rating'});
disp(data)
```

```
      ID             Date            Rating
    _______    ________________    _______

    {'ABC'}    {'17-Feb-2015'}    {'AA' }
    {'ABC'}    {'6-Jul-2017' }    {'A'  }
    {'LMN'}    {'12-Aug-2014'}    {'B'  }
    {'LMN'}    {'9-Nov-2015' }    {'CCC'}
    {'LMN'}    {'7-Sep-2016' }    {'D'  }
    {'XYZ'}    {'14-May-2013'}    {'BB' }
    {'XYZ'}    {'21-Jun-2016'}    {'BBB'}
```

The `transprob` function understands that this panel-data format indicates the dates when a new rating is assigned to a given company. `transprob` assumes that such ratings remain unchanged, unless a subsequent row explicitly indicates a rating change. For example, for company `'ABC'`, `transprob` understands that the `'A'` rating is unchanged for any date after `'6-Jul-2017'` (indefinitely).

Compute Transition Matrix and Transition Counts

The `transprob` function returns a transition probability matrix as the primary output. There are also optional outputs that contain additional information for how many transitions occurred. For more information, see `transprob` for information on the optional outputs for both the `'cohort'` and the `'duration'` methods.

For illustration purposes, this example allows you to pick the `StartYear` (limited to `2014` or `2015` for this example) and the `EndYear` (`2016` or `2017`). This example also uses the `hDisplayTransitions` helper function (see the Local Functions on page 8-117 section) to format the transitions information for ease of reading.

```
StartYear = 2014 ;
EndYear = 2017 ;
startDate = datetime(StartYear,12,31,'Locale','en_US');
```

```
endDate = datetime(EndYear,12,31,'Locale','en_US');
RatingLabels = ["AAA","AA","A","BBB","BB","B","CCC","D"];

[tm,st,it] = transprob(data,'startDate',startDate,'endDate',endDate,'algorithm','cohort','labels
```

The transition probabilities of the TransMat output indicate the probability of migrating between ratings. The probabilities are expressed in %, that is, they are multiplied by 100.

```
hDisplayTransitions(tm,RatingLabels,"Transition Matrix")
```

```
Transition Matrix

           AAA    AA     A      BBB    BB    B    CCC     D
           ___    __    ___    ___    __    _    ___    ___

    AAA    100     0      0      0     0    0      0      0
    AA       0    50     50      0     0    0      0      0
    A        0     0    100      0     0    0      0      0
    BBB      0     0      0    100     0    0      0      0
    BB       0     0      0     50    50    0      0      0
    B        0     0      0      0     0    0    100      0
    CCC      0     0      0      0     0    0      0    100
    D        0     0      0      0     0    0      0    100
```

The transition counts are stored in the sampleTotals optional output and indicate how many transitions occurred between ratings for the entire sample (that is, all companies).

```
hDisplayTransitions(st.totalsMat,RatingLabels,"Transition counts, all companies")
```

```
Transition counts, all companies

           AAA    AA    A    BBB    BB    B    CCC    D
           ___    __    _    ___    __    _    ___    _

    AAA     0     0     0     0     0     0     0     0
    AA      0     1     1     0     0     0     0     0
    A       0     0     0     0     0     0     0     0
    BBB     0     0     0     1     0     0     0     0
    BB      0     0     0     1     1     0     0     0
    B       0     0     0     0     0     0     1     0
    CCC     0     0     0     0     0     0     0     1
    D       0     0     0     0     0     0     0     1
```

The third output of transprob is idTotals that contains information about transitions at an ID level, company by company (in the same order that the companies appear in the input data).

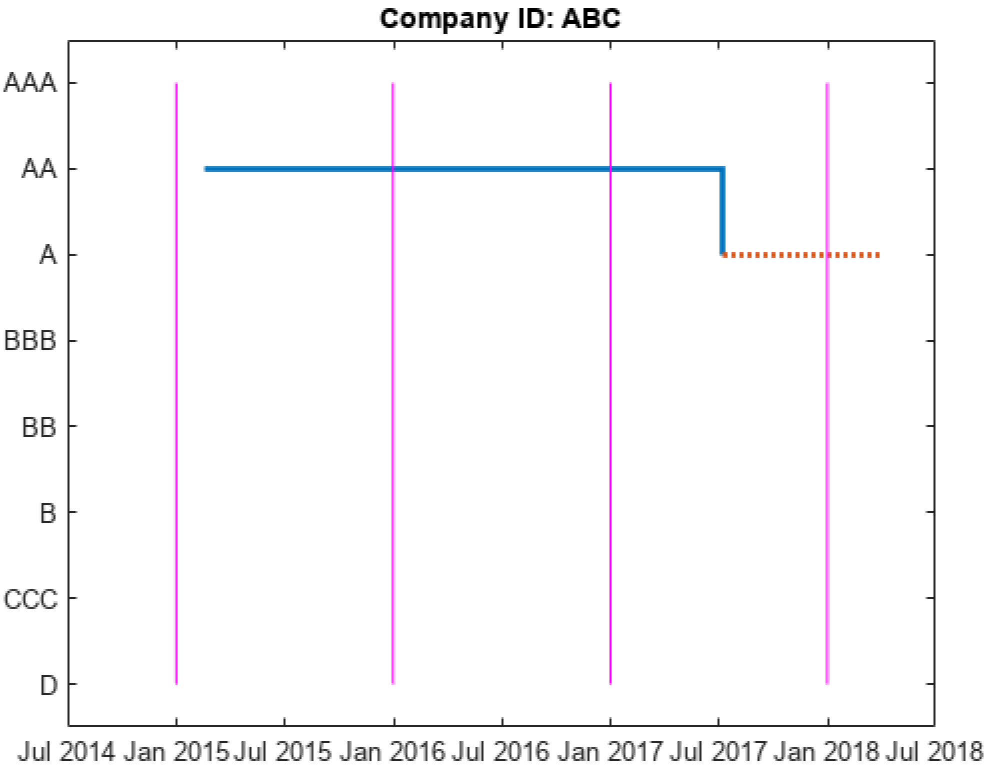
Select a company to display the transition counts and a corresponding visualization of the transitions. The hPlotTransitions helper function (see the Local Functions on page 8-117 section) shows the transitions history for a company.

```
CompanyID = ABC ;
UniqueIDs = unique(data.ID,'stable');
[~,CompanyIndex] = ismember(CompanyID,UniqueIDs);
hDisplayTransitions(it(CompanyIndex).totalsMat,RatingLabels,strcat("Transition counts, company II
```

```
Transition counts, company ID: ABC
```

```
              AAA    AA    A    BBB    BB    B    CCC    D
              ___    __    _    ___    __    _    ___    _

       AAA     0     0     0     0     0     0     0     0
       AA      0     1     1     0     0     0     0     0
       A       0     0     0     0     0     0     0     0
       BBB     0     0     0     0     0     0     0     0
       BB      0     0     0     0     0     0     0     0
       B       0     0     0     0     0     0     0     0
       CCC     0     0     0     0     0     0     0     0
       D       0     0     0     0     0     0     0     0
```

```
hPlotTransitions(CompanyID,startDate,endDate,data,RatingLabels)
```



To understand how `transprob` handles data when the first observed date is after the start date of the analysis, or whose last observed date occurs before the end date of the analysis, consider the following example. For company `'ABC'` suppose that the analysis has a start date of `31-Dec-2014` and end date of `31-Dec-2017`. There are only two transitions reported for this company for that analysis time window. The first observation for `'ABC'` happened on `17-Feb-2015`. So the `31-Dec-2015` snapshot is the first time the company is observed. By `31-Dec-2016`, the company remained in the original `'AA'` rating. By `31-Dec-2017`, a downgrade to `'A'` is recorded. Consistent with this, the transition counts show one transition from `'AA'` to `'AA'` (from the end of 2015 to the end of 2016), and one transition from `'AA'` to `'A'` (from the end of 2016 to the end of 2017). The plot shows the last rating as a dotted red line to emphasize that the last rating in the data is extrapolated indefinitely into the future. There is no extrapolation into the past; the company's history is ignored until a company rating is known for an entire transition period (`31-Dec-2015` through `31-Dec-2016` in the case of `'ABC'`).

Compute Transition Matrix Containing NR (Not Rated) Rating

Consider a different sample data containing only a single company `'DEF'`. The data contains transitions of company `'DEF'` from `'A'` to `'NR'` rating and a subsequent transition from `'NR'` to `'BBB'`.

```
dataNR = {'DEF','17-Mar-2011','A';
    'DEF','24-Mar-2014','NR';
    'DEF','26-Sep-2016','BBB'};
dataNR = cell2table(dataNR,'VariableNames',{'ID','Date','Rating'});
disp(dataNR)
```

```
      ID             Date            Rating
    _______    _______________    _______

    {'DEF'}    {'17-Mar-2011'}    {'A'  }
    {'DEF'}    {'24-Mar-2014'}    {'NR' }
    {'DEF'}    {'26-Sep-2016'}    {'BBB'}
```

`transprob` treats `'NR'` as another rating. The transition matrix below shows the estimated probability of transitioning into and out of `'NR'`.

```
StartYearNR = 2010;
EndYearNR = 2018;
startDateNR = datetime(StartYearNR,12,31,'Locale','en_US');
endDateNR = datetime(EndYearNR,12,31,'Locale','en_US');
CompanyID_NR = "DEF";

RatingLabelsNR = ["AAA","AA","A","BBB","BB","B","CCC","D","NR"];

[tmNR,~,itNR] = transprob(dataNR,'startDate',startDateNR,'endDate',endDateNR,'algorithm','cohort
hDisplayTransitions(tmNR,RatingLabelsNR,"Transition Matrix")
```

```
Transition Matrix

           AAA    AA       A       BBB    BB      B     CCC     D       NR
           ___    ___    ______    ___    ___    ___    ___    ___    ______

    AAA    100      0         0      0      0      0      0      0         0
    AA       0    100         0      0      0      0      0      0         0
    A        0      0    66.667      0      0      0      0      0    33.333
    BBB      0      0         0    100      0      0      0      0         0
    BB       0      0         0      0    100      0      0      0         0
    B        0      0         0      0      0    100      0      0         0
    CCC      0      0         0      0      0      0    100      0         0
    D        0      0         0      0      0      0      0    100         0
    NR       0      0         0     50      0      0      0      0        50
```

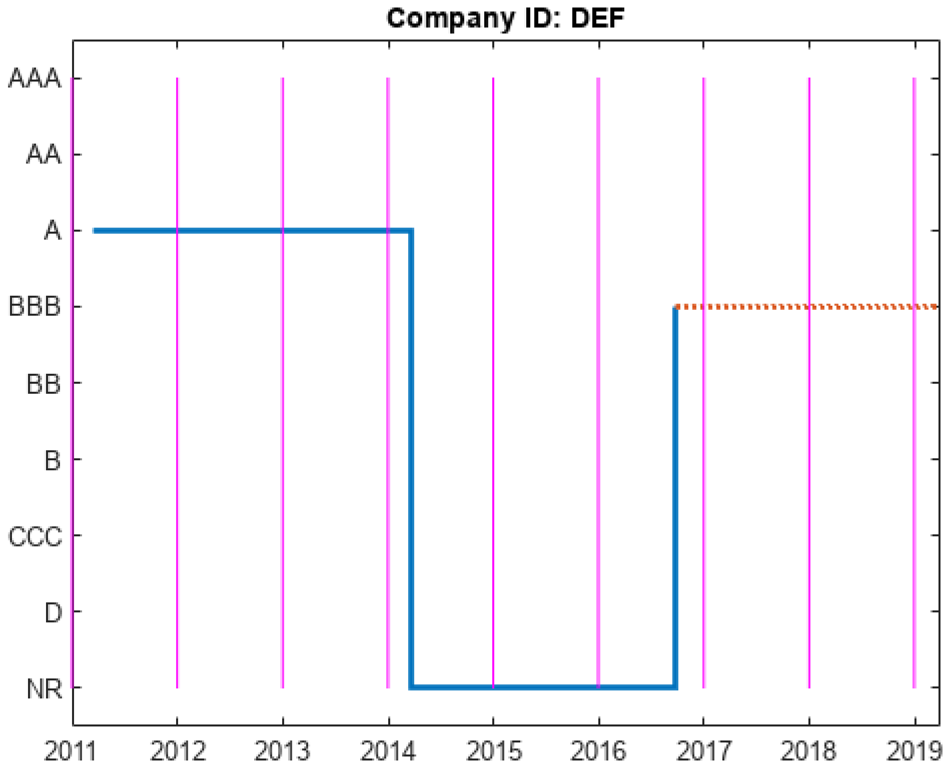
Display the transition counts and corresponding visualization of the transitions.

```
hDisplayTransitions(itNR.totalsMat,RatingLabelsNR,strcat("Transition counts, company ID: ",Compar
```

```
Transition counts, company ID: DEF

           AAA    AA    A    BBB    BB    B    CCC    D    NR
```

```
              ___     __     _     ___     __     _     ___     _     __

    AAA       0      0      0      0      0      0      0      0      0
    AA        0      0      0      0      0      0      0      0      0
    A         0      0      2      0      0      0      0      0      1
    BBB       0      0      0      2      0      0      0      0      0
    BB        0      0      0      0      0      0      0      0      0
    B         0      0      0      0      0      0      0      0      0
    CCC       0      0      0      0      0      0      0      0      0
    D         0      0      0      0      0      0      0      0      0
    NR        0      0      0      1      0      0      0      0      1
```

```
hPlotTransitions(CompanyID_NR,startDateNR,endDateNR,dataNR,RatingLabelsNR)
```



To remove the `'NR'` from the transition matrix, use the `'excludeLabels'` name-value input argument in `transprob`. The list of labels to exclude may or may not be specified in the name-value pair argument `labels`. For example, both `RatingLabels` and `RatingLabelsNR` generate the same output from `transprob`.

```
[tmNR,stNR,itNR] = transprob(dataNR,'startDate',startDateNR,'endDate',endDateNR,'algorithm','coho
hDisplayTransitions(tmNR,RatingLabels,"Transition Matrix")
```

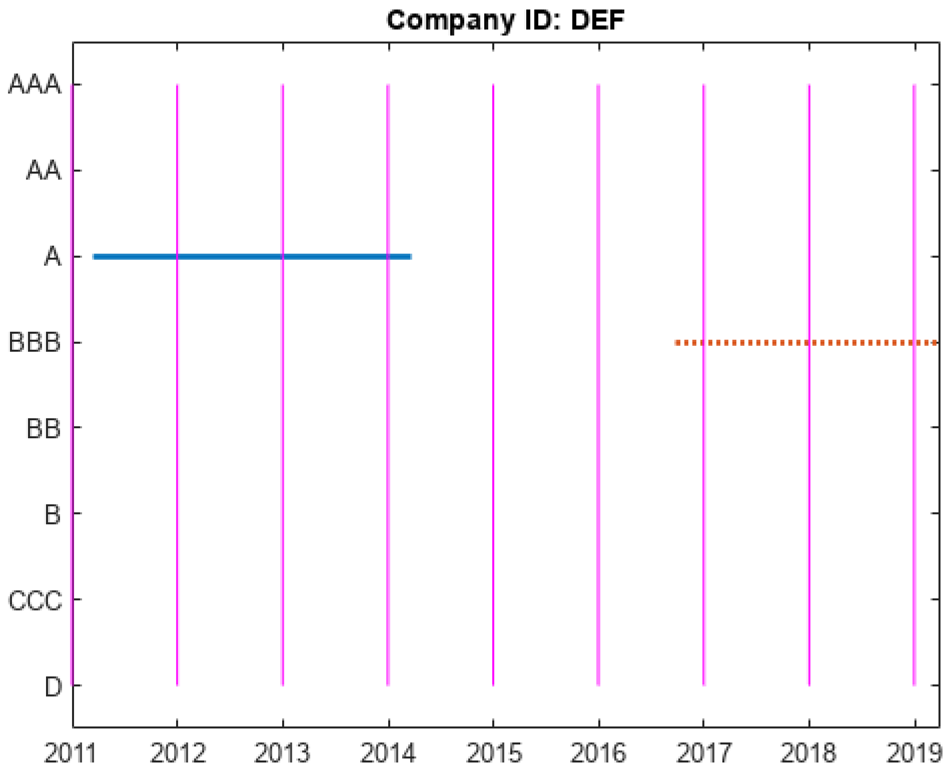
```
Transition Matrix

              AAA    AA      A     BBB    BB      B     CCC     D
              ___    ___    ___    ___    ___    ___    ___    ___

    AAA       100      0      0      0      0      0      0      0
    AA       0    100      0      0      0      0      0      0
    A        0      0    100      0      0      0      0      0
    BBB      0      0      0    100      0      0      0      0
    BB       0      0      0      0    100      0      0      0
    B        0      0      0      0      0    100      0      0
    CCC      0      0      0      0      0      0    100      0
    D        0      0      0      0      0      0      0    100
```

Display the transition counts and corresponding visualization of the transitions.

```
hDisplayTransitions(itNR.totalsMat,RatingLabels,strcat("Transition counts, company ID: ",CompanyI
```

```
Transition counts, company ID: DEF

           AAA    AA    A    BBB    BB    B    CCC    D
           ___    __    _    ___    __    _    ___    _

    AAA     0     0     0     0     0     0     0     0
    AA      0     0     0     0     0     0     0     0
    A       0     0     2     0     0     0     0     0
    BBB     0     0     0     2     0     0     0     0
    BB      0     0     0     0     0     0     0     0
    B       0     0     0     0     0     0     0     0
    CCC     0     0     0     0     0     0     0     0
    D       0     0     0     0     0     0     0     0
```

```
hPlotTransitions(CompanyID_NR,startDateNR,endDateNR,dataNR,RatingLabels)
```



Consistent with the previous plot, the transition counts still show two transitions from `'A'` to `'A'` (from the end of 2012 to the end of 2014), and two transitions from `'BBB'` to `'BBB'` (from the end of 2017 to the end of 2019).

However, different from the previous plot, specifying `'NR'` using the `'excludeLabels'` name-value input argument of `transprob` removes any transitions into and out of the `'NR'` rating.

Local Functions

```
function hDisplayTransitions(TransitionsData,RatingLabels,Title)
% Helper function to format transition information outputs

TransitionsAsTable = array2table(TransitionsData,...
   'VariableNames',RatingLabels,'RowNames',RatingLabels);

fprintf('\n%s\n\n',Title)
disp(TransitionsAsTable)

end

function hPlotTransitions(CompanyID,startDate,endDate,data,RatingLabels)
% Helper function to visualize transitions between ratings

   Ind = string(data.ID)==CompanyID;
   DatesOriginal = datetime(data.Date(Ind),'Locale','en_US');
   RatingsOriginal = categorical(data.Rating(Ind),flipud(RatingLabels(:)),flipud(RatingLabels(:))

   stairs(DatesOriginal,RatingsOriginal,'LineWidth',2)
   hold on;

   % Indicate rating extrapolated into the future (arbitrarily select 91
   % days after endDate as the last date on the plot)
   endDateExtrap = endDate+91;
   if endDateExtrap>DatesOriginal(end)
      DatesExtrap = [DatesOriginal(end); endDateExtrap];
      RatingsExtrap = [RatingsOriginal(end); RatingsOriginal(end)];
      stairs(DatesExtrap,RatingsExtrap,'LineWidth',2,'LineStyle',':')
   end
   hold off;

   % Add lines to indicate the snapshot dates
   % transprob uses 1 as the default for 'snapsPerYear', hardcoded here for simplicity
   % The call to cfdates generates the exact same snapshot dates that transprob uses
   snapsPerYear = 1;
   snapDates = cfdates(startDate-1,endDate,snapsPerYear)';
   yLimits = ylim;
   for ii=1:length(snapDates)
      line([snapDates(ii) snapDates(ii)],yLimits,'Color','m')
   end
   title(strcat("Company ID: ",CompanyID))
end
```

Impute Missing Data in the Credit Scorecard Workflow Using the k-Nearest Neighbors Algorithm

This example shows how to perform imputation of missing data in the credit scorecard workflow using the k-nearest neighbors (kNN) algorithm.

The kNN algorithm is a nonparametric method used for classification and regression. In both cases, the input consists of the k-closest training examples in the feature space. The output depends on whether kNN is used for classification or regression. In kNN classification, an object is classified by a plurality vote of its neighbors, and the object is assigned to the class most common among its k-nearest neighbors. In kNN regression, the output is the average of the values of k-nearest neighbors. For more information on the kNN algorithm, see `fitcknn`.

For additional information on alternative approaches for "treating" missing data, see “Credit Scorecard Modeling with Missing Values” on page 8-56.

Impute Missing Data Using kNN Algorithm

Use the `dataMissing` data set to impute missing values for the `CustAge` (numeric) and `ResStatus` (categorical) predictors.

```
load CreditCardData.mat
disp(head(dataMissing));
```

```
    CustID    CustAge    TmAtAddress     ResStatus     EmpStatus    CustIncome    TmWBank    Othe
    ______    _______    ___________    ___________    _________    __________    _______    ____

      1          53          62         <undefined>    Unknown        50000         55         Ye
      2          61          22         Home Owner     Employed       52000         25         Ye
      3          47          30         Tenant         Employed       37000         61         No
      4         NaN          75         Home Owner     Employed       53000         20         Ye
      5          68          56         Home Owner     Employed       53000         14         Ye
      6          65          13         Home Owner     Employed       48000         59         Ye
      7          34          32         Home Owner     Unknown        32000         26         Ye
      8          50          57         Other          Employed       51000         33         No
```

In this example, the `'CustID'` and `'status'` columns are removed in the imputation process as those are the `id` and `response` values respectively. Alternatively, you can choose to leave the `'status'` column in.

```
dataToImpute = dataMissing(:,setdiff(dataMissing.Properties.VariableNames,...
    {'CustID','status'},'stable'));
```

Create dummy variables for all categorical predictors so that the kNN algorithm can compute the Euclidean distances.

```
dResStatus = dummyvar(dataToImpute.ResStatus);
dEmpStatus = dummyvar(dataToImpute.EmpStatus);
dOtherCC = dummyvar(dataToImpute.OtherCC);
```

`'k'` in the kNN algorithm is based on feature similarity. Choosing the right value of `'k'` is a process called parameter tuning, which is important for greater accuracy. There is no physical way to determine the "best" value for `'k'`, so you have to try a few values before settling on one. Small values of `'k'` can be noisy and subject to the effects of outliers. Larger values of `'k'` have smoother decision boundaries, which mean lower variance but increased bias.

For the purpose of this example, choose `'k'` as the square root of the number of samples in the data set. This is a generally accepted value for `'k'`. Choose a value of `'k'` that is odd in order to break a tie between two classes of data.

```
numObs = height(dataToImpute);
k = round(sqrt(numObs));
if ~mod(k,2)
    k = k+1;
end
```

Get the missing values from the `CustAge` and `ResStatus` predictors.

```
missingResStatus = ismissing(dataToImpute.ResStatus);
missingCustAge = ismissing(dataToImpute.CustAge);
```

Next, follow these steps:

- Modify the dataset to incorporate the dummy variables.
- Call the `fitcknn` function to create a k-nearest neighbor classifier.
- Call the `predict` method on that class to predict the imputed values.

```
custAgeToImpute = dataToImpute;
custAgeToImpute.HomeOwner = dResStatus(:,1);
custAgeToImpute.Tenant = dResStatus(:,2);
custAgeToImpute.Employed = dEmpStatus(:,1);
custAgeToImpute.HasOtherCC = dOtherCC(:,2);
custAgeToImpute = removevars(custAgeToImpute, 'ResStatus');
custAgeToImpute = removevars(custAgeToImpute, 'EmpStatus');
custAgeToImpute = removevars(custAgeToImpute, 'OtherCC');

knnCustAge = fitcknn(custAgeToImpute, 'CustAge', 'NumNeighbors', k, 'Standardize',true);
imputedCustAge = predict(knnCustAge,custAgeToImpute(missingCustAge,:));

resStatusToImpute = dataToImpute;
resStatusToImpute.Employed = dEmpStatus(:,1);
resStatusToImpute.HasOtherCC = dOtherCC(:,2);
resStatusToImpute = removevars(resStatusToImpute, 'EmpStatus');
resStatusToImpute = removevars(resStatusToImpute, 'OtherCC');

knnResStatus = fitcknn(resStatusToImpute, 'ResStatus', 'NumNeighbors', k, 'Standardize', true);
imputedResStatus = predict(knnResStatus,resStatusToImpute(missingResStatus,:));
```

Compare Imputed Data to Original Data

Create a new data set with the imputed data.

```
knnImputedData = dataMissing;
knnImputedData.CustAge(missingCustAge) = imputedCustAge;
knnImputedData.ResStatus(missingResStatus) = imputedResStatus;
disp(knnImputedData(5:10,:));
```

```
    CustID    CustAge    TmAtAddress     ResStatus     EmpStatus    CustIncome    TmWBank    Othe
    ______    _______    ___________    __________    _________    __________    _______    ____

      5         68           56         Home Owner    Employed       53000         14         Ye
      6         65           13         Home Owner    Employed       48000         59         Ye
      7         34           32         Home Owner    Unknown        32000         26         Ye
      8         50           57         Other         Employed       51000         33         No
```

```
     9       50          10        Tenant        Unknown        52000        25        Yes
    10       49          30        Home Owner    Unknown        53000        23        Yes
```

```
disp(knnImputedData(find(missingCustAge,5),:));
```

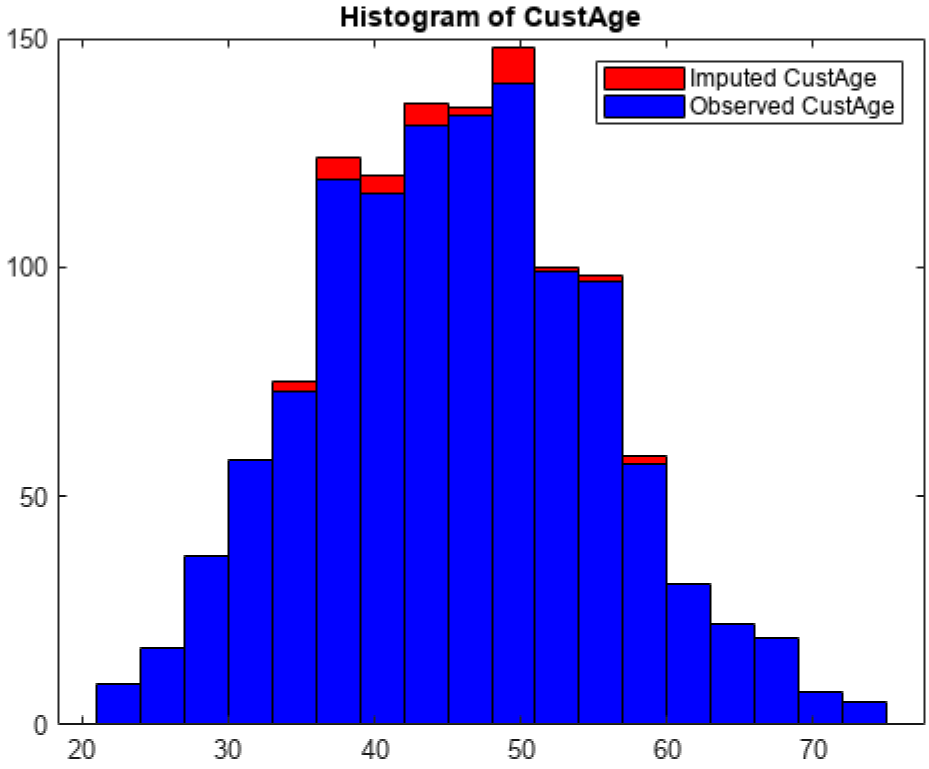
```
    CustID    CustAge    TmAtAddress     ResStatus     EmpStatus    CustIncome    TmWBank    Othe
    ______    _______    ___________    __________    _________    __________    _______    ____

       4        52           75         Home Owner    Employed       53000         20        Yes
      19        45           14         Home Owner    Employed       51000         11        Yes
     138        41           31         Other         Employed       41000          2        Yes
     165        37           21         Home Owner    Unknown        38000         70        No 
     207        48           38         Home Owner    Employed       48000         12        No 
```

```
disp(knnImputedData(find(missingResStatus,5),:));
```

```
    CustID    CustAge    TmAtAddress     ResStatus     EmpStatus    CustIncome    TmWBank    Othe
    ______    _______    ___________    __________    _________    __________    _______    ____

       1        53           62         Tenant        Unknown        50000         55        Yes
      22        51           13         Tenant        Employed       35000         33        Yes
      33        46            8         Home Owner    Unknown        32000         26        Yes
      47        52           56         Tenant        Employed       56000         79        Yes
     103        64           49         Tenant        Employed       50000         35        Yes
```

Plot a histogram of the predictor values before and after imputation.

```
Predictor = Customer Age ▾;
f1 = figure;
ax1 = axes(f1);
histogram(ax1,knnImputedData.(Predictor),'FaceColor','red','FaceAlpha',1);
hold on
histogram(ax1,dataMissing.(Predictor),'FaceColor','blue','FaceAlpha',1);
legend(strcat("Imputed ", Predictor), strcat("Observed ", Predictor));
title(strcat("Histogram of ", Predictor));
```



Create Credit Scorecard Model Using New Imputed Data

Use the imputed data to create the `creditscorecard` object, and then use `autobinning`, `fitmodel`, and `formatpoints` to create a credit scorecard model.

```
sc = creditscorecard(knnImputedData,'IDVar','CustID');
sc = autobinning(sc);
[sc,mdl] = fitmodel(sc,'display','off');
sc = formatpoints(sc,'PointsOddsAndPDO',[500 2 50]);
PointsInfo = displaypoints(sc);
disp(PointsInfo);
```

```
      Predictors                Bin                Points
    ______________    _____________________    ______

    {'CustAge'   }    {'[-Inf,33)'        }    53.675
    {'CustAge'   }    {'[33,37)'          }    56.983
    {'CustAge'   }    {'[37,40)'          }    57.721
    {'CustAge'   }    {'[40,45)'          }    67.063
    {'CustAge'   }    {'[45,48)'          }    78.319
    {'CustAge'   }    {'[48,51)'          }    79.494
    {'CustAge'   }    {'[51,58)'          }    81.157
    {'CustAge'   }    {'[58,Inf]'         }    97.315
    {'CustAge'   }    {'<missing>'        }       NaN
    {'ResStatus' }    {'Tenant'           }    63.012
    {'ResStatus' }    {'Home Owner'       }     72.35
    {'ResStatus' }    {'Other'            }    92.434
    {'ResStatus' }    {'<missing>'        }       NaN
    {'EmpStatus' }    {'Unknown'           }    58.892
    {'EmpStatus' }    {'Employed'          }     86.83
    {'EmpStatus' }    {'<missing>'         }       NaN
    {'CustIncome'}    {'[-Inf,29000)'      }    30.304
    {'CustIncome'}    {'[29000,33000)'     }    56.365
    {'CustIncome'}    {'[33000,35000)'     }    67.971
    {'CustIncome'}    {'[35000,40000)'     }    70.136
    {'CustIncome'}    {'[40000,42000)'     }    70.936
    {'CustIncome'}    {'[42000,47000)'     }    82.196
    {'CustIncome'}    {'[47000,Inf]'       }    96.405
    {'CustIncome'}    {'<missing>'         }       NaN
    {'TmWBank'   }    {'[-Inf,12)'         }    50.966
    {'TmWBank'   }    {'[12,23)'           }    60.975
    {'TmWBank'   }    {'[23,45)'           }    61.778
    {'TmWBank'   }    {'[45,71)'           }    93.007
    {'TmWBank'   }    {'[71,Inf]'          }    133.39
    {'TmWBank'   }    {'<missing>'         }       NaN
    {'OtherCC'   }    {'No'                }    50.765
    {'OtherCC'   }    {'Yes'               }    75.649
    {'OtherCC'   }    {'<missing>'         }       NaN
    {'AMBalance' }    {'[-Inf,558.88)'     }    89.765
    {'AMBalance' }    {'[558.88,1254.28)'  }    63.097
    {'AMBalance' }    {'[1254.28,1597.44)'}    59.725
    {'AMBalance' }    {'[1597.44,Inf]'     }    49.184
    {'AMBalance' }    {'<missing>'         }       NaN
```

Calculate Scores and Probability of Default for New Applicants

Create a data set of `'new customers'` and then calculate the scores and probabilities of default.

```
dataNewCustomers = dataMissing(1:20,1:end-1);
disp(head(dataNewCustomers));
```

```
    CustID    CustAge    TmAtAddress     ResStatus     EmpStatus    CustIncome    TmWBank    Othe
    ______    _______    ___________    ___________    _________    __________    _______    ____

      1         53           62         <undefined>    Unknown        50000         55         Ye
      2         61           22         Home Owner     Employed       52000         25         Ye
      3         47           30         Tenant         Employed       37000         61         No
      4        NaN           75         Home Owner     Employed       53000         20         Ye
      5         68           56         Home Owner     Employed       53000         14         Ye
      6         65           13         Home Owner     Employed       48000         59         Ye
      7         34           32         Home Owner     Unknown        32000         26         Ye
      8         50           57         Other          Employed       51000         33         No
```

Perform the same preprocessing on the `'new customers'` data as on the training data.

```
dResStatusNewCustomers = dummyvar(dataNewCustomers.ResStatus);
dEmpStatusNewCustomers = dummyvar(dataNewCustomers.EmpStatus);
dOtherCCNewCustomers = dummyvar(dataNewCustomers.OtherCC);

dataNewCustomersCopy = dataNewCustomers;
dataNewCustomersCopy.HomeOwner = dResStatusNewCustomers(:,1);
dataNewCustomersCopy.Tenant = dResStatusNewCustomers(:,2);
dataNewCustomersCopy.Employed = dEmpStatusNewCustomers(:,1);
dataNewCustomersCopy.HasOtherCC = dOtherCCNewCustomers(:,2);
dataNewCustomersCopy = removevars(dataNewCustomersCopy, 'ResStatus');
dataNewCustomersCopy = removevars(dataNewCustomersCopy, 'EmpStatus');
dataNewCustomersCopy = removevars(dataNewCustomersCopy, 'OtherCC');
```

Predict the missing data in the scoring data set with the same imputation model as before.

```
missingCustAgeNewCustomers = isnan(dataNewCustomers.CustAge);
missingResStatusNewCustomers = ismissing(dataNewCustomers.ResStatus);
imputedCustAgeNewCustomers = round(predict(knnCustAge, dataNewCustomersCopy(missingCustAgeNewCust
imputedResStatusNewCustomers = predict(knnResStatus, dataNewCustomersCopy(missingResStatusNewCust
dataNewCustomers.CustAge(missingCustAgeNewCustomers) = imputedCustAgeNewCustomers;
dataNewCustomers.ResStatus(missingResStatusNewCustomers) = imputedResStatusNewCustomers;
```

Use `score` to calculate scores of new customers.

```
[scores, points] = score(sc, dataNewCustomers);
disp(scores);
```

```
  531.2201
  553.4261
  505.1671
  563.1321
  552.6226
  584.6546
  445.1156
  516.8917
  524.9965
  507.6668
  498.2255
  539.4057
  516.4594
  491.6344
  566.1685
  486.8248
  476.0595
  469.5488
  550.2850
  511.0285
```

```
disp(points);
```

```
    CustAge    ResStatus    EmpStatus    CustIncome    TmWBank    OtherCC    AMBalance
    _______    _________    _________    __________    _______    _______    _________

    81.157      63.012       58.892        96.405      93.007     75.649      63.097  
    97.315       72.35        86.83        96.405      61.778     75.649      63.097  
    78.319      63.012        86.83        70.136      93.007     50.765      63.097  
    81.157       72.35        86.83        96.405      60.975     75.649      89.765  
    97.315       72.35        86.83        96.405      60.975     75.649      63.097  
    97.315       72.35        86.83        96.405      93.007     75.649      63.097  
    56.983       72.35       58.892        56.365      61.778     75.649      63.097  
    79.494      92.434        86.83        96.405      61.778     50.765      49.184  
    79.494      63.012       58.892        96.405      61.778     75.649      89.765  
    79.494       72.35       58.892        96.405      61.778     75.649      63.097  
    81.157      63.012       58.892        67.971      61.778     75.649      89.765  
    79.494      92.434       58.892        82.196      60.975     75.649      89.765  
    97.315       72.35       58.892        96.405      50.966     50.765      89.765  
    67.063      92.434       58.892        70.936      61.778     50.765      89.765  
    78.319      92.434        86.83        82.196      60.975     75.649      89.765  
    56.983       72.35        86.83        70.136      61.778     75.649      63.097  
    57.721      63.012        86.83        67.971      61.778     75.649      63.097  
    53.675       72.35        86.83        30.304      60.975     75.649      89.765  
```

```
             78.319       72.35        86.83        96.405      50.966     75.649      89.765
             81.157      92.434       58.892        82.196      60.975     75.649      59.725
```

Impute Missing Data in the Credit Scorecard Workflow Using the Random Forest Algorithm

This example shows how to perform imputation of missing data in the credit scorecard workflow using the random forest algorithm.

Random forests are an ensemble learning method for classification or regression that operates by constructing a multitude of decision trees at training time and obtaining the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees. Random forests correct for the tendency of decision trees to overfit to the training set. For more information on the random forest algorithm, see `fitrensemble` and `fitcensemble`.

For additional information on alternative approaches for "treating" missing data, see “Credit Scorecard Modeling with Missing Values” on page 8-56.

Impute Missing Data Using Random Forest Algorithm

Use the `dataMissing` data set to impute missing values for the `CustAge` (numeric) and `ResStatus` (categorical) predictors.

```
load CreditCardData.mat
disp(head(dataMissing));
```

```
    CustID     CustAge     TmAtAddress      ResStatus       EmpStatus     CustIncome     TmWBank     Othe
    ______     _______     ___________     ___________     _________     __________     _______     ____

      1          53            62          <undefined>     Unknown         50000          55         Ye
      2          61            22          Home Owner      Employed        52000          25         Ye
      3          47            30          Tenant          Employed        37000          61         No
      4         NaN            75          Home Owner      Employed        53000          20         Ye
      5          68            56          Home Owner      Employed        53000          14         Ye
      6          65            13          Home Owner      Employed        48000          59         Ye
      7          34            32          Home Owner      Unknown         32000          26         Ye
      8          50            57          Other           Employed        51000          33         No
```

Remove the `'CustID'` and `'status'` columns in the imputation process as these are the `id` and `response` values respectively. Alternatively, you can choose to leave the `'status'` column in.

```
dataToImpute = dataMissing(:,setdiff(dataMissing.Properties.VariableNames,...
    {'CustID','status'},'stable'));

rfImputedData = dataMissing;
```

Because multiple predictors contain missing data, turn on the `'Surrogate'` flag when you create the decision tree template.

```
rng('default');
tmp = templateTree('Surrogate','on','Reproducible',true);
```

Next, use the `fitrensemble` and `fitcensemble` functions, which return the trained regression and classification ensemble model objects contain the results of boosting 100 regression and classification trees using `LSBoost`, respectively.

```
missingCustAge = ismissing(dataToImpute.CustAge);
% Fit ensemble of regression learners
```

```
rfCustAge = fitrensemble(dataToImpute,'CustAge','Method','Bag',...
    'NumLearningCycles',200,'Learners',tmp,'CategoricalPredictors',...
    {'ResStatus','EmpStatus','OtherCC'});
rfImputedData.CustAge(missingCustAge) = predict(rfCustAge,...
    dataToImpute(missingCustAge,:));

missingResStatus = ismissing(dataToImpute.ResStatus);
% Fit ensemble of classification learners
rfResStatus = fitcensemble(dataToImpute,'ResStatus','Method','Bag',...
    'NumLearningCycles',200,'Learners',tmp,'CategoricalPredictors',...
    {'EmpStatus','OtherCC'});
rfImputedData.ResStatus(missingResStatus) = predict(rfResStatus,...
    dataToImpute(missingResStatus,:));

% Optionally, round the age to the nearest integer
rfImputedData.CustAge = round(rfImputedData.CustAge);
```

Compare Imputed Data to Original Data

```
disp(rfImputedData(5:10,:));
```

```
    CustID    CustAge    TmAtAddress     ResStatus     EmpStatus    CustIncome    TmWBank    Othe
    ______    _______    ___________    __________    _________    __________    _______    ____

      5         68           56         Home Owner    Employed       53000         14        Ye
      6         65           13         Home Owner    Employed       48000         59        Ye
      7         34           32         Home Owner    Unknown        32000         26        Ye
      8         50           57         Other         Employed       51000         33        No
      9         50           10         Tenant        Unknown        52000         25        Ye
     10         49           30         Home Owner    Unknown        53000         23        Ye
```

```
disp(rfImputedData(find(missingCustAge,5),:));
```

```
    CustID    CustAge    TmAtAddress     ResStatus     EmpStatus    CustIncome    TmWBank    Othe
    ______    _______    ___________    __________    _________    __________    _______    ____

      4         55           75         Home Owner    Employed       53000         20        Ye
     19         54           14         Home Owner    Employed       51000         11        Ye
    138         52           31         Other         Employed       41000          2        Ye
    165         46           21         Home Owner    Unknown        38000         70        No
    207         52           38         Home Owner    Employed       48000         12        No
```

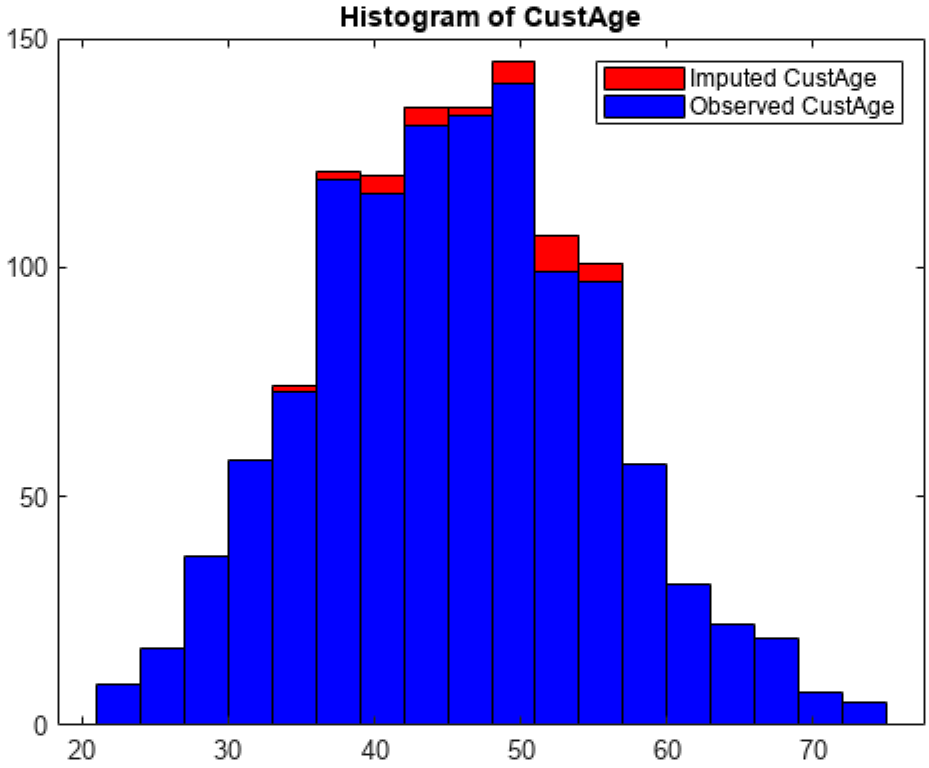
```
disp(rfImputedData(find(missingResStatus,5),:));
```

```
    CustID    CustAge    TmAtAddress     ResStatus     EmpStatus    CustIncome    TmWBank    Othe
    ______    _______    ___________    __________    _________    __________    _______    ____

      1         53           62         Tenant        Unknown        50000         55        Ye
     22         51           13         Home Owner    Employed       35000         33        Ye
     33         46            8         Home Owner    Unknown        32000         26        Ye
     47         52           56         Tenant        Employed       56000         79        Ye
    103         64           49         Home Owner    Employed       50000         35        Ye
```

Plot a histogram of the predictor values before and after imputation.

```
Predictor = Customer Age ;
f1 = figure;
ax1 = axes(f1);
```

```
histogram(ax1,rfImputedData.(Predictor),'FaceColor','red','FaceAlpha',1);
hold on
histogram(ax1,dataMissing.(Predictor),'FaceColor','blue','FaceAlpha',1);
legend(strcat("Imputed ", Predictor), strcat("Observed ", Predictor));
title(strcat("Histogram of ", Predictor));
```



Create Credit Scorecard Model Using New Imputed Data

Use the imputed data to create the `creditscorecard` object, and then use `autobinning`, `fitmodel`, and `formatpoints` to create a credit scorecard model.

```
sc = creditscorecard(rfImputedData,'IDVar','CustID');
sc = autobinning(sc);
[sc,mdl] = fitmodel(sc,'display','off');
sc = formatpoints(sc,'PointsOddsAndPDO',[500 2 50]);
PointsInfo = displaypoints(sc);
disp(PointsInfo);
```

```
      Predictors               Bin              Points
    ______________    _____________________    ______

    {'CustAge'   }    {'[-Inf,33)'        }    54.313
    {'CustAge'   }    {'[33,37)'          }    57.145
    {'CustAge'   }    {'[37,40)'          }     59.04
    {'CustAge'   }    {'[40,46)'          }    68.806
    {'CustAge'   }    {'[46,51)'          }    78.204
    {'CustAge'   }    {'[51,58)'          }    81.041
    {'CustAge'   }    {'[58,Inf]'         }    96.395
    {'CustAge'   }    {'<missing>'        }       NaN
    {'ResStatus' }    {'Tenant'           }    62.768
    {'ResStatus' }    {'Home Owner'       }    72.621
    {'ResStatus' }    {'Other'            }    92.228
    {'ResStatus' }    {'<missing>'        }       NaN
    {'EmpStatus' }    {'Unknown'          }    58.839
    {'EmpStatus' }    {'Employed'         }    86.897
    {'EmpStatus' }    {'<missing>'        }       NaN
    {'CustIncome'}    {'[-Inf,29000)'     }    29.765
    {'CustIncome'}    {'[29000,33000)'    }    56.167
    {'CustIncome'}    {'[33000,35000)'    }    67.926
    {'CustIncome'}    {'[35000,40000)'    }    70.119
    {'CustIncome'}    {'[40000,42000)'    }     70.93
    {'CustIncome'}    {'[42000,47000)'    }    82.337
    {'CustIncome'}    {'[47000,Inf]'      }    96.733
    {'CustIncome'}    {'<missing>'        }       NaN
    {'TmWBank'   }    {'[-Inf,12)'        }    51.023
    {'TmWBank'   }    {'[12,23)'          }    61.005
    {'TmWBank'   }    {'[23,45)'          }    61.806
    {'TmWBank'   }    {'[45,71)'          }     92.95
    {'TmWBank'   }    {'[71,Inf]'         }    133.22
    {'TmWBank'   }    {'<missing>'        }       NaN
    {'OtherCC'   }    {'No'               }    50.796
    {'OtherCC'   }    {'Yes'              }    75.644
    {'OtherCC'   }    {'<missing>'        }       NaN
    {'AMBalance' }    {'[-Inf,558.88)'    }    89.941
    {'AMBalance' }    {'[558.88,1254.28)' }    63.018
    {'AMBalance' }    {'[1254.28,1597.44)'}    59.613
    {'AMBalance' }    {'[1597.44,Inf]'    }    48.972
    {'AMBalance' }    {'<missing>'        }       NaN
```

Calculate Scores and Probability of Default for New Customers

Create a data set of `'new customers'` and then calculate the scores and probabilities of default.

```
dataNewCustomers = dataMissing(1:20,1:end-1);
disp(head(dataNewCustomers));
```

```
    CustID    CustAge    TmAtAddress     ResStatus     EmpStatus    CustIncome    TmWBank    Othe
    ______    _______    ___________    ___________    _________    __________    _______    ____

      1          53          62         <undefined>    Unknown        50000         55         Ye
      2          61          22         Home Owner     Employed       52000         25         Ye
      3          47          30         Tenant         Employed       37000         61         No
      4         NaN          75         Home Owner     Employed       53000         20         Ye
      5          68          56         Home Owner     Employed       53000         14         Ye
      6          65          13         Home Owner     Employed       48000         59         Ye
      7          34          32         Home Owner     Unknown        32000         26         Ye
      8          50          57         Other          Employed       51000         33         No
```

Predict missing data in the scoring data set with the same imputation model as before.

```
missingCustAgeNewCustomers = isnan(dataNewCustomers.CustAge);
missingResStatusNewCustomers = ismissing(dataNewCustomers.ResStatus);
imputedCustAgeNewCustomers = round(predict(rfCustAge, dataNewCustomers(missingCustAgeNewCustomers
imputedResStatusNewCustomers = predict(rfResStatus, dataNewCustomers(missingResStatusNewCustomers
dataNewCustomers.CustAge(missingCustAgeNewCustomers) = imputedCustAgeNewCustomers;
dataNewCustomers.ResStatus(missingResStatusNewCustomers) = imputedResStatusNewCustomers;
```

Use `score` to calculate the scores of the new customers.

```
[scores, points] = score(sc, dataNewCustomers);
disp(scores);
```

```
  530.9936
  553.1144
  504.7522
  563.8821
  552.3131
  584.2581
  445.2402
  515.6361
  523.9354
  506.8645
  497.9661
  538.1986
  516.3480
  493.3467
  566.2568
  487.2501
  477.0996
  470.1861
  553.9004
  510.7086
```

```
disp(points);
```

```
    CustAge    ResStatus    EmpStatus    CustIncome    TmWBank    OtherCC    AMBalance
    _______    _________    _________    __________    _______    _______    _________

    81.041      62.768       58.839        96.733        92.95    75.644      63.018  
    96.395      72.621       86.897        96.733       61.806    75.644      63.018  
    78.204      62.768       86.897        70.119        92.95    50.796      63.018  
    81.041      72.621       86.897        96.733       61.005    75.644      89.941  
    96.395      72.621       86.897        96.733       61.005    75.644      63.018  
    96.395      72.621       86.897        96.733        92.95    75.644      63.018  
    57.145      72.621       58.839        56.167       61.806    75.644      63.018  
    78.204      92.228       86.897        96.733       61.806    50.796      48.972  
    78.204      62.768       58.839        96.733       61.806    75.644      89.941  
    78.204      72.621       58.839        96.733       61.806    75.644      63.018  
    81.041      62.768       58.839        67.926       61.806    75.644      89.941  
    78.204      92.228       58.839        82.337       61.005    75.644      89.941  
    96.395      72.621       58.839        96.733       51.023    50.796      89.941  
    68.806      92.228       58.839         70.93       61.806    50.796      89.941  
    78.204      92.228       86.897        82.337       61.005    75.644      89.941  
    57.145      72.621       86.897        70.119       61.806    75.644      63.018  
     59.04      62.768       86.897        67.926       61.806    75.644      63.018  
    54.313      72.621       86.897        29.765       61.005    75.644      89.941  
    81.041      72.621       86.897        96.733       51.023    75.644      89.941  
    81.041      92.228       58.839        82.337       61.005    75.644      59.613  
```

Treat Missing Data in a Credit Scorecard Workflow Using MATLAB® fillmissing

This example shows a workflow to gather missing data, manually treat the training data, develop a new `creditscorecard`, and treat new data before scoring using the MATLAB® `fillmissing`.

The advantage of this method is that you can use all the options available in `fillmissing` to fill missing data, as well as other MATLAB functionality such as `standardizeMissing` and features for the treatment of outliers. In this approach, note that you must ensure that the treatment of the training data and the treatment of any new data set that requires scoring must be the same.

Alternatively, after you create a `creditscorecard` object, you can use the `fillmissing` function for the `creditscorecard` object to fill missing values. For additional information on alternative approaches for "treating" missing data, see “Credit Scorecard Modeling with Missing Values” on page 8-56.

The `dataMissing` table in the `CreditCardData.mat` file has two predictors with missing values — `CustAge` and `ResStatus`.

```
load CreditCardData.mat
head(dataMissing)
```

```
    CustID    CustAge    TmAtAddress     ResStatus     EmpStatus    CustIncome    TmWBank    Othe
    ______    _______    ___________    ___________    _________    __________    _______    ____

      1          53          62         <undefined>    Unknown        50000         55        Ye
      2          61          22         Home Owner     Employed       52000         25        Ye
      3          47          30         Tenant         Employed       37000         61        No
      4         NaN          75         Home Owner     Employed       53000         20        Ye
      5          68          56         Home Owner     Employed       53000         14        Ye
      6          65          13         Home Owner     Employed       48000         59        Ye
      7          34          32         Home Owner     Unknown        32000         26        Ye
      8          50          57         Other          Employed       51000         33        No
```

First, analyze the missing data information using the untreated training data.

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `dataMissing` that contains missing values. Set the `'BinMissingData'` argument for `creditscorecard` to `true` to explicitly report information on missing values. Then apply automatic binning using `autobinning`.

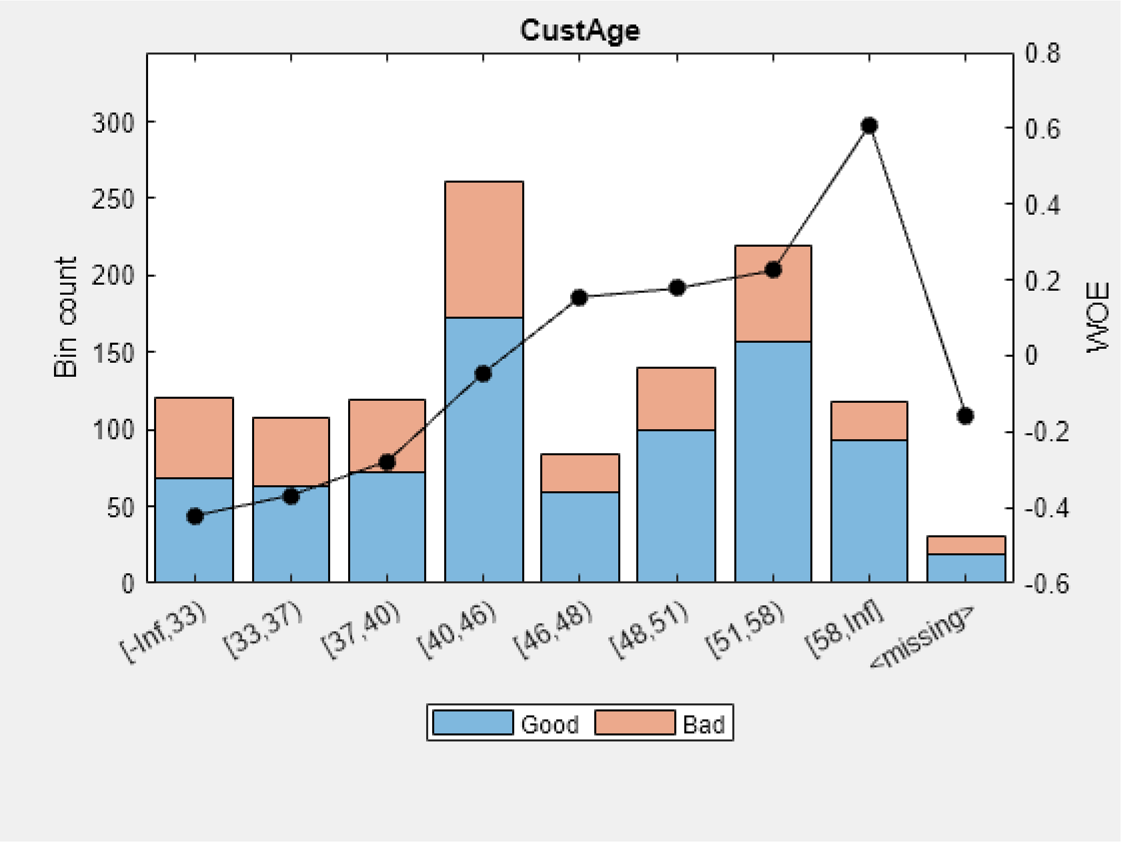
```
sc = creditscorecard(dataMissing,'IDVar','CustID','BinMissingData',true);
sc = autobinning(sc);
```

The bin information and bin plots for predictors that have missing data both show a `<missing>` bin at the end. The two predictors with missing values in this data set are `CustAge` and `ResStatus`.

```
bi = bininfo(sc,'CustAge');
disp(bi)
```

```
         Bin          Good    Bad     Odds        WOE       InfoValue
    ______________    ____    ___    ______    _______    __________

    {'[-Inf,33)'}      69     52     1.3269    -0.42156     0.018993
    {'[33,37)'  }      63     45        1.4    -0.36795     0.012839
    {'[37,40)'  }      72     47     1.5319     -0.2779    0.0079824
    {'[40,46)'  }    172      89    1.9326    -0.04556     0.0004549
    {'[46,48)'  }     59      25      2.36     0.15424     0.0016199
    {'[48,51)'  }     99      41    2.4146     0.17713     0.0035449
    {'[51,58)'  }    157      62    2.5323     0.22469     0.0088407
    {'[58,Inf]' }     93      25      3.72     0.60931      0.032198
    {'<missing>'}     19      11    1.7273    -0.15787    0.00063885
    {'Totals'   }    803     397    2.0227         NaN      0.087112
```

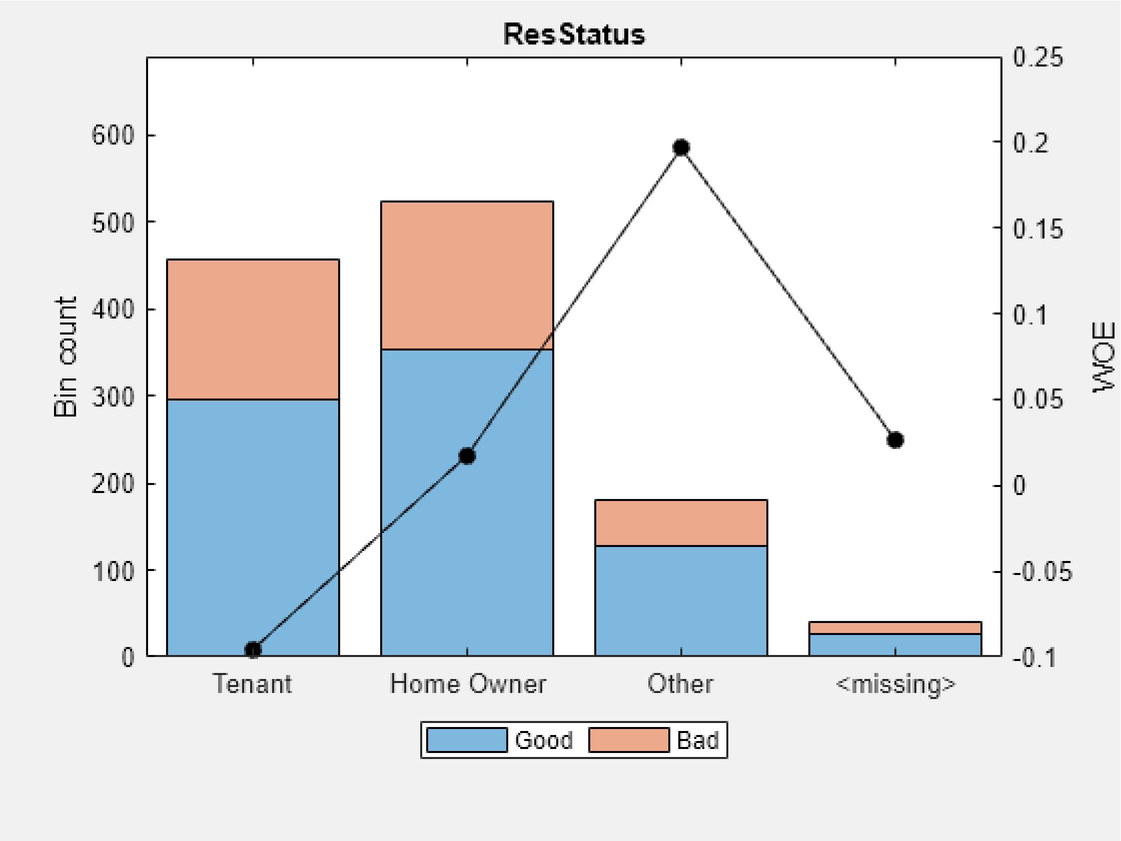
```
plotbins(sc,'CustAge')
```



```
bi = bininfo(sc,'ResStatus');
disp(bi)
```

```
         Bin            Good    Bad     Odds        WOE       InfoValue 
    ______________      ____    ___    ______    _________    __________

    {'Tenant'     }     296     161    1.8385    -0.095463     0.0035249
    {'Home Owner'}      352     171    2.0585     0.017549    0.00013382
    {'Other'      }     128      52    2.4615      0.19637     0.0055808
    {'<missing>' }       27      13    2.0769     0.026469    2.3248e-05
    {'Totals'     }     803     397    2.0227          NaN     0.0092627
```

```
plotbins(sc,'ResStatus')
```



The missing bin can be left as is, although a common alternative is to treat the missing values. Note that treating the missing values must be done with care because it changes the data and can introduce bias.

To treat missing values, you can apply different criteria. This example follows a straightforward approach to replace missing observations with the most common or typical value in the data distribution, which is the value of `mode` for the data. For this example, the `mode` happens to have a similar WOE value as the original `<missing>` bin. The similarity in values is favorable because similar WOE values means similar points in a scorecard.

For `CustAge`, bin `4` is the bin with the most observations and the `mode` value of the original data is `43`.

```
modeCustAge = mode(dataMissing.CustAge);
disp(modeCustAge)
```

```
    43
```

The WOE value of the `<missing>` bin is similar to the WOE value of bin `4`. Therefore, replacing the missing values in `CustAge` with the value of `mode` is reasonable.

To treat the data, create a copy of the data and fill the missing values.

```
dataTreated = dataMissing;
dataTreated.CustAge = fillmissing(dataTreated.CustAge,'constant',modeCustAge);
```

For `ResStatus`, the value of `'Home Owner'` is the value of the `mode` of the data, and the WOE value of the `<missing>` bin is closest to that of the `'Home Owner'` bin.

```
modeResStatus = mode(dataMissing.ResStatus);
disp(modeResStatus)
```

```
     Home Owner
```

Use MATLAB® `fillmissing` to replace the missing data with `'Home Owner'`.

```
dataTreated.ResStatus = fillmissing(dataTreated.ResStatus,'constant',string(modeResStatus));
```

The treated data set now has no missing values.

```
disp(any(any(ismissing(dataTreated))))
```

```
   0
```

Using the treated data set, apply the typical `creditscorecard` workflow. First, create a `creditscorecard` object with the treated data and then apply automatic binning.

```
scTreated = creditscorecard(dataTreated,'IDVar','CustID');
scTreated = autobinning(scTreated);
```

Compare the bin information of the untreated data for `CustAge` with the bin information of the treated data for `CustAge`.

```
bi = bininfo(sc,'CustAge');
disp(bi)
```

```
         Bin           Good    Bad     Odds        WOE       InfoValue 
    _____________      ____    ___    ______    _________    __________

    {'[-Inf,33)'}       69      52    1.3269     -0.42156      0.018993
    {'[33,37)'  }       63      45       1.4     -0.36795      0.012839
    {'[37,40)'  }       72      47    1.5319      -0.2779     0.0079824
    {'[40,46)'  }      172      89    1.9326     -0.04556     0.0004549
    {'[46,48)'  }       59      25      2.36      0.15424     0.0016199
    {'[48,51)'  }       99      41    2.4146      0.17713     0.0035449
    {'[51,58)'  }      157      62    2.5323      0.22469     0.0088407
    {'[58,Inf]' }       93      25      3.72      0.60931      0.032198
    {'<missing>'}       19      11    1.7273     -0.15787    0.00063885
    {'Totals'   }      803     397    2.0227          NaN      0.087112
```

```
biTreated = bininfo(scTreated,'CustAge');
disp(biTreated)
```

```
         Bin           Good    Bad     Odds        WOE       InfoValue
    _____________      ____    ___    ______    _________    _________

    {'[-Inf,33)'}       69      52    1.3269     -0.42156     0.018993
    {'[33,37)'  }       63      45       1.4     -0.36795     0.012839
    {'[37,40)'  }       72      47    1.5319      -0.2779    0.0079824
    {'[40,45)'  }      156      86     1.814     -0.10891    0.0024345
    {'[45,48)'  }       94      39    2.4103      0.17531    0.0033002
    {'[48,58)'  }      256     103    2.4854      0.20603      0.01223
    {'[58,Inf]' }       93      25      3.72      0.60931     0.032198
    {'Totals'   }      803     397    2.0227          NaN     0.089977
```

The first few bins are the same, but the treatment of missing values influences the binning results, starting with the bin where the missing data is placed. You can further explore your binning results using `autobinning` with a different algorithm or you can manually modify the bins using `modifybins`.

For `ResStatus`, the results for the treated data look similar to the initial results, except for the higher counts in the `'Home Owner'` bin due to the treatment. For a categorical variable with more categories (or levels), an automatic algorithm can find category groups and the results can show more differences for before and after the treatment.

```
bi = bininfo(sc,'ResStatus');
disp(bi)
```

```
         Bin            Good    Bad      Odds         WOE        InfoValue 
    ________________    ____    ___    ______    _________    __________

    {'Tenant'      }    296     161    1.8385    -0.095463     0.0035249
    {'Home Owner'}      352     171    2.0585     0.017549    0.00013382
    {'Other'       }    128      52    2.4615      0.19637     0.0055808
    {'<missing>' }       27      13    2.0769     0.026469    2.3248e-05
    {'Totals'      }    803     397    2.0227          NaN     0.0092627
```

```
biTreated = bininfo(scTreated,'ResStatus');
disp(biTreated)
```

```
         Bin            Good    Bad      Odds         WOE        InfoValue 
    ________________    ____    ___    ______    _________    __________

    {'Tenant'      }    296     161    1.8385    -0.095463     0.0035249
    {'Home Owner'}      379     184    2.0598     0.018182    0.00015462
    {'Other'       }    128      52    2.4615      0.19637     0.0055808
    {'Totals'      }    803     397    2.0227          NaN     0.0092603
```

Fit the logistic model, scale the points, and display the final scorecard.

```
[scTreated, mdl] = fitmodel(scTreated,'Display','off');
scTreated = formatpoints(scTreated,'PointsOddsAndPDO',[500 2 50]);
ScPoints = displaypoints(scTreated);
disp(ScPoints)
```

```
      Predictors                 Bin               Points
    ________________    ______________________    ______

    {'CustAge'    }    {'[-Inf,33)'          }    53.507
    {'CustAge'    }    {'[33,37)'            }    55.798
    {'CustAge'    }    {'[37,40)'            }    59.646
    {'CustAge'    }    {'[40,45)'            }    66.868
    {'CustAge'    }    {'[45,48)'            }    79.013
    {'CustAge'    }    {'[48,58)'            }    80.326
    {'CustAge'    }    {'[58,Inf]'           }    97.559
    {'CustAge'    }    {'<missing>'          }       NaN
    {'ResStatus'  }    {'Tenant'             }    62.161
    {'ResStatus'  }    {'Home Owner'         }    73.305
    {'ResStatus'  }    {'Other'              }    90.777
    {'ResStatus'  }    {'<missing>'          }       NaN
    {'EmpStatus'  }    {'Unknown'            }    58.846
    {'EmpStatus'  }    {'Employed'           }    86.887
    {'EmpStatus'  }    {'<missing>'          }       NaN
    {'CustIncome'}    {'[-Inf,29000)'     }    29.906
    {'CustIncome'}    {'[29000,33000)'    }    56.219
    {'CustIncome'}    {'[33000,35000)'    }    67.938
    {'CustIncome'}    {'[35000,40000)'    }    70.123
    {'CustIncome'}    {'[40000,42000)'    }    70.931
    {'CustIncome'}    {'[42000,47000)'    }      82.3
    {'CustIncome'}    {'[47000,Inf]'      }    96.647
    {'CustIncome'}    {'<missing>'        }       NaN
    {'TmWBank'   }    {'[-Inf,12)'        }     51.05
    {'TmWBank'   }    {'[12,23)'          }    61.018
    {'TmWBank'   }    {'[23,45)'          }    61.818
    {'TmWBank'   }    {'[45,71)'          }    92.921
    {'TmWBank'   }    {'[71,Inf]'         }    133.14
    {'TmWBank'   }    {'<missing>'        }       NaN
    {'OtherCC'   }    {'No'               }    50.806
    {'OtherCC'   }    {'Yes'              }    75.642
    {'OtherCC'   }    {'<missing>'        }       NaN
    {'AMBalance' }    {'[-Inf,558.88)'    }    89.788
    {'AMBalance' }    {'[558.88,1254.28)' }    63.088
    {'AMBalance' }    {'[1254.28,1597.44)'}    59.711
    {'AMBalance' }    {'[1597.44,Inf]'    }    49.157
    {'AMBalance' }    {'<missing>'        }       NaN
```

The new scorecard does not know that the data was treated, hence it assigns NaNs to the <missing> bins. If you need to score a new data set and it contains missing data, by default, the `score` function sets the points to NaN. To further explore the handling of missing data, take a few rows from the original data as test data and introduce some missing data.

```
tdata = dataTreated(11:14,mdl.PredictorNames); % Keep only the predictors retained in the model
% Set some missing values
tdata.CustAge(1) = NaN;
tdata.ResStatus(2) = '<undefined>';
tdata.EmpStatus(3) = '<undefined>';
tdata.CustIncome(4) = NaN;
disp(tdata)
```

```
    CustAge     ResStatus       EmpStatus      CustIncome    TmWBank    OtherCC    AMBalance
    _______    ___________    ___________    __________    _______    _______    _________

      NaN      Tenant         Unknown          34000         44         Yes         119.8 
       48      <undefined>    Unknown          44000         14         Yes        403.62 
       65      Home Owner     <undefined>      48000          6         No         111.88 
       44      Other          Unknown            NaN         35         No         436.41 
```

Score the new data and see how points are set to NaN, which leads to NaN scores.

```
[Scores,Points] = score(scTreated,tdata);
disp(Scores)
```

```
   NaN
   NaN
   NaN
   NaN
```

```
disp(Points)
```

```
    CustAge    ResStatus    EmpStatus    CustIncome    TmWBank    OtherCC    AMBalance
    _______    _________    _________    __________    _______    _______    _________
```

```
       NaN      62.161       58.846        67.938      61.818      75.642      89.788
    80.326         NaN       58.846          82.3      61.018      75.642      89.788
    97.559      73.305          NaN        96.647       51.05      50.806      89.788
    66.868      90.777       58.846           NaN      61.818      50.806      89.788
```

To assign points to missing data, one possibility is to use the name-value pair argument `'Missing'` in `formatpoints` to choose how to assign points to missing values.

Use the `'MinPoints'` option for the `'Missing'` argument. This option assigns the minimum number of possible points in the scorecard to the missing data. In this example, the minimum number of possible points for `CustIncome` is `29.906`, so the last row in the table gets `29.906` points for the missing `CustIncome` value.

```
scTreated = formatpoints(scTreated,'Missing','MinPoints');
[Scores,Points] = score(scTreated,tdata);
disp(Scores)
```

```
  469.7003
  510.0812
  518.0013
  448.8099
```

```
disp(Points)
```

```
    CustAge    ResStatus    EmpStatus    CustIncome    TmWBank    OtherCC    AMBalance
    _______    _________    _________    __________    _______    _______    _________

    53.507      62.161       58.846        67.938      61.818      75.642      89.788
    80.326      62.161       58.846          82.3      61.018      75.642      89.788
    97.559      73.305       58.846        96.647       51.05      50.806      89.788
    66.868      90.777       58.846        29.906      61.818      50.806      89.788
```

However, for predictors treated in the training data, such as `CustAge`, the effect of the `'Missing'` argument is inconsistent with the treatment of the training data. For example, for `CustAge`, the first observation gets `53.507` points for the missing value, yet if the new data were "treated," and the missing value for `CustAge` were replaced with the `mode` of the training data (age of `43`), this observation falls in the `[40,45)` bin and receives `66.868` points.

Therefore, before scoring, data sets must be treated the same way the training data was treated. The use of the `'Missing'` argument is still important to assign points for untreated predictors and the treated predictors receive points in a way that is consistent with the way the model was developed.

```
tdataTreated = tdata;
tdataTreated.CustAge = fillmissing(tdataTreated.CustAge,'constant',modeCustAge);
tdataTreated.ResStatus = fillmissing(tdataTreated.ResStatus,'constant',string(modeResStatus));
disp(tdataTreated)
```

```
    CustAge    ResStatus      EmpStatus     CustIncome    TmWBank    OtherCC    AMBalance
    _______    __________    ___________    __________    _______    _______    _________

      43       Tenant        Unknown          34000      44         Yes       119.8
      48       Home Owner    Unknown          44000      14         Yes      403.62
      65       Home Owner    <undefined>      48000       6         No       111.88
      44       Other         Unknown            NaN      35         No       436.41
```

```
[Scores,Points] = score(scTreated,tdataTreated);
disp(Scores)
```

```
  483.0606
  521.2249
  518.0013
  448.8099
```

```
disp(Points)
```

```
    CustAge    ResStatus    EmpStatus    CustIncome    TmWBank    OtherCC    AMBalance
    _______    _________    _________    __________    _______    _______    _________

    66.868      62.161       58.846        67.938      61.818     75.642      89.788  
    80.326      73.305       58.846          82.3      61.018     75.642      89.788  
    97.559      73.305       58.846        96.647       51.05     50.806      89.788  
    66.868      90.777       58.846        29.906      61.818     50.806      89.788  
```

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 9-21
- "Estimation of Some Technology Stock Betas" on page 9-27

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 $Z_1, ..., Z_m$ 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 $H_1, ..., H_m$, 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 Z_k is assumed to have independent identically distributed multivariate normal residual errors $Z_k - H_k\, b$ 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 $z_1, ..., z_m$ of the random variables $Z_1, ..., Z_m$. 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 $z_1, \dots , z_m$.

Given the multivariate normal model to characterize residual errors in the regression model, the log-likelihood function is

$$L(z_1, \dots, z_m;\ b,\ C) = \frac{1}{2}mn\log(2\pi) + \frac{1}{2}m\log(\det(C))$$
$$+\frac{1}{2}\sum_{k=1}^{m}(z_k - H_k b)^T C^{-1}(z_k - H_k b)\,.$$

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 $t = 0, 1, \dots$ with the superscript notation $b^{(t)}$ and $C^{(t)}$.

Given initial estimates $b^{(0)}$ and $C^{(0)}$ 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 $t = 0, 1, \dots$.

Special Case of Multiple Linear Regression Model

The special case mentioned in “Maximum Likelihood Estimation” on page 9-3 occurs if $n = 1$ so that the sequence of observations is a sequence of scalar observations. This model is known as a multiple linear regression model. In this case, the covariance matrix 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 $b^{(1)}$ and $C^{(1)}$.

Least-Squares Regression

Another simplification of the general model is called least-squares regression. If $b^{(0)} = 0$ and $C^{(0)} = I$, then $b^{(1)}$ and $C^{(1)}$ from the two-stage iterative process are least-squares estimates for b and 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 $z_1, ..., z_m$. 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 $H_k = I$ for $i = 1, ..., m$ so that b is an estimate for the mean and C is an estimate of the covariance of the collection of observations $z_1, ..., z_m$.

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 b^{ML} and 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, ...

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

$$s^2\left(\widehat{\theta}_i\right) = \left(I^{-1}\left(\widehat{\theta}_i\right)\right)_{ii}$$

for i = 1, ..., 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

$$\widehat{\theta}_i \pm 1.96 s\left(\widehat{\theta}_i\right)$$

for i = 1, ..., q.

Error ellipses at a level-of-significance α ε [0, 1] for the parameter estimates satisfy the inequality

$$\left(\theta - \widehat{\theta}\right)^T I\left(\widehat{\theta}\right)\left(\theta - \widehat{\theta}\right) \leq \chi^2_{1-\alpha,\, q}$$

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

`mvnrmle` | `mvnrstd` | `mvnrfish` | `mvnrobj` | `ecmmvnrmle` | `ecmmvnrstd` | `ecmmvnrfish` | `ecmmvnrobj` | `ecmlsrmle` | `ecmlsrobj` | `ecmmvnrstd` | `ecmmvnrfish` | `ecmnmle` | `ecmnstd` | `ecmnfish` | `ecmnhess` | `ecmnobj` | `convert2sur` | `ecmninit`

Related Examples

- “Maximum Likelihood Estimation with Missing Data” on page 9-7
- “Multivariate Normal Regression Types” on page 9-13
- “Valuation with Missing Data” on page 9-26
- “Portfolios with Missing Data” on page 9-21

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 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 ignorable 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 x_k represent the missing values in z_k and y_k represent the observed values. Define a permutation matrix 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 $b^{(t)}$ and $C^{(t)}$.

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 $\widehat{C}$, each observation has moments

$$E[Z_k] = H_k\widehat{b}$$

and

$$cov(Z_k) = H_k\widehat{C}H_k{}^T$$

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 X_k and observed values $Y_k = y_k$, you can form conditional estimates for any subcollection of random variables within a given observation. Thus, given estimates $E[\ Z_k\ ]$ and $cov(Z_k)$ based on the parameter estimates, you can create conditional estimates

$$E[X_k|y_k]$$

and

$$cov(X_k|y_k)$$

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

$$X_k \sim N(E[X_k|y_k],\ cov(X_k|y_k))\,.$$

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

`mvnrmle` | `mvnrstd` | `mvnrfish` | `mvnrobj` | `ecmmvnrmle` | `ecmmvnrstd` | `ecmmvnrfish` | `ecmmvnrobj` | `ecmlsrmle` | `ecmlsrobj` | `ecmmvnrstd` | `ecmmvnrfish` | `ecmnmle` | `ecmnstd` | `ecmnfish` | `ecmnhess` | `ecmnobj` | `convert2sur` | `ecmninit`

Related Examples

- “Multivariate Normal Regression” on page 9-2
- “Multivariate Normal Regression Types” on page 9-13
- “Valuation with Missing Data” on page 9-26
- “Portfolios with Missing Data” on page 9-21

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 9-11
- “Multivariate Normal Regression With Missing Data” on page 9-11
- “Least-Squares Regression With Missing Data” on page 9-11
- “Multivariate Normal Parameter Estimation With Missing Data” on page 9-12

Additional support functions are also provided, see “Support Functions” on page 9-12.

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

$$\text{Data(k, :)} = z_k^T$$

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

`mvnrmle` | `mvnrstd` | `mvnrfish` | `mvnrobj` | `ecmmvnrmle` | `ecmmvnrstd` | `ecmmvnrfish` | `ecmmvnrobj` | `ecmlsrmle` | `ecmlsrobj` | `ecmmvnrstd` | `ecmmvnrfish` | `ecmnmle` | `ecmnstd` | `ecmnfish` | `ecmnhess` | `ecmnobj` | `convert2sur` | `ecmninit`

Related Examples

- “Multivariate Normal Regression” on page 9-2
- “Multivariate Normal Regression Types” on page 9-13
- “Valuation with Missing Data” on page 9-26
- “Portfolios with Missing Data” on page 9-21

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 using `mvnrmle`:

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

Estimate standard errors using `mvnrstd`:

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

Multivariate Normal Regression With Missing Data

Estimate parameters using `ecmmvnrmle`:

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

Estimate standard errors using `ecmmvnrstd`:

```
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 using `mvnrmle`:

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

Estimate standard errors using `mvnrstd`:

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

Least-Squares Regression With Missing Data

Estimate parameters using `ecmlsrmle`:

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

Estimate standard errors using `ecmmvnrstd`:

```
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 using `mvnrmle`:

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

Estimate standard errors using `mvnrstd`:

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

Covariance-Weighted Least Squares With Missing Data

Estimate parameters using `ecmlsrmle`:

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

Estimate standard errors using `ecmmvnrstd`:

```
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 using `mvnrmle`:

```
[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 using `mvnrstd`:

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

Feasible Generalized Least Squares With Missing Data

Estimate parameters using ecmlsrmle:

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

Estimate standard errors using ecmmvnrstd:

```
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 a SUR design using `convert2sur`:

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

Estimate parameters using `mvnrmle`:

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

Estimate standard errors using `mvnrstd`:

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

Seemingly Unrelated Regression With Missing Data

Form a SUR design using `convert2sur`:

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

Estimate parameters using `ecmmvnrmle`:

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

Estimate standard errors using `ecmmvnrstd`:

```
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 using `ecmnmle`:

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

Estimate standard errors using `ecmnstd`:

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

See Also

`mvnrmle` | `mvnrstd` | `mvnrfish` | `mvnrobj` | `ecmmvnrmle` | `ecmmvnrstd` | `ecmmvnrfish` | `ecmmvnrobj` | `ecmlsrmle` | `ecmlsrobj` | `ecmmvnrstd` | `ecmmvnrfish` | `ecmnmle` | `ecmnstd` | `ecmnfish` | `ecmnhess` | `ecmnobj` | `convert2sur` | `ecmninit`

Related Examples

- “Multivariate Normal Regression” on page 9-2
- “Maximum Likelihood Estimation with Missing Data” on page 9-7
- “Valuation with Missing Data” on page 9-26
- “Portfolios with Missing Data” on page 9-21

Troubleshooting Multivariate Normal Regression

This section provides a few pointers to handle various technical and operational difficulties that might occur.

Biased Estimates

If samples are ignored, the number of samples used in the estimation is less than `NumSamples`. Clearly the actual number of samples used must be sufficient to obtain estimates. In addition, although the model parameters `Parameters` (or mean estimates `Mean`) are unbiased maximum likelihood estimates, the residual covariance estimate `Covariance` is biased. To convert to an unbiased covariance estimate, multiply `Covariance` by

$$\text{Count}/(\text{Count}-1),$$

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

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

and the least-squares regression functions require

$$\text{Count} \times \text{NumSeries} \le \text{NumParams},$$

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

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

Second, they must have enough nonmissing values to converge. Third, they must have a nondegenerate covariance matrix.

Although some necessary and sufficient conditions can be found in the references, general conditions for existence and uniqueness of solutions in the missing-data case, do not exist. Nonconvergence is usually due to an ill-conditioned covariance matrix estimate, which is discussed in greater detail in “Nonconvergence” on page 9-19.

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.

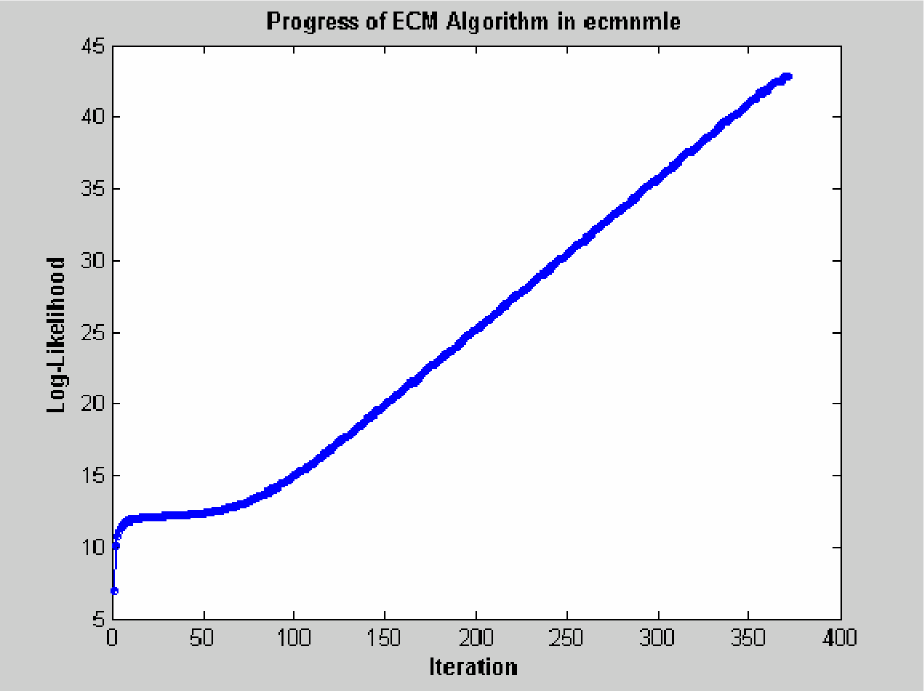
Other improvised approaches are possible although most approaches are problem-dependent. In particular, for mean and covariance estimation, the estimation function `ecmnmle` uses a function `ecmninit` to obtain an initial estimate.

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

`mvnrmle` | `mvnrstd` | `mvnrfish` | `mvnrobj` | `ecmmvnrmle` | `ecmmvnrstd` | `ecmmvnrfish` | `ecmmvnrobj` | `ecmlsrmle` | `ecmlsrobj` | `ecmmvnrstd` | `ecmmvnrfish` | `ecmnmle` | `ecmnstd` | `ecmnfish` | `ecmnhess` | `ecmnobj` | `convert2sur` | `ecmninit`

Related Examples

- “Multivariate Normal Regression” on page 9-2
- “Maximum Likelihood Estimation with Missing Data” on page 9-7
- “Valuation with Missing Data” on page 9-26
- “Portfolios with Missing Data” on page 9-21

Portfolios with Missing Data

This example shows 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 six hardware and six software companies. The example estimates the mean and covariance matrix for these stocks, forms efficient frontiers with both a naïve approach and the ECM approach, and then compares results.

Load the data file.

```
load ecmtechdemo.mat
```

This data 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.

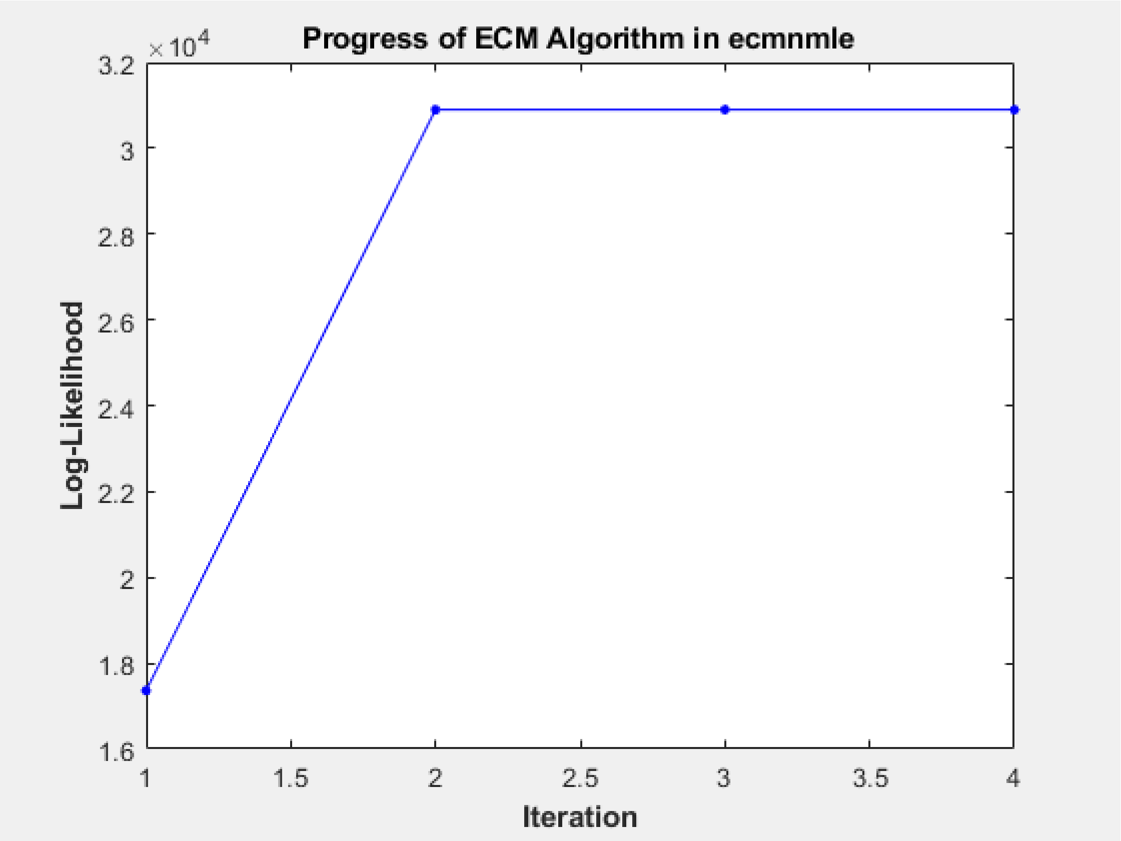
The time period for the data extends from April 19, 2000 to April 18, 2005. The sixth stock in `Assets` is Google (GOOG), which started trading on August 19, 2004. So, all returns before August 20, 2004 are missing and represented as `NaN`s. Also, Amazon (AMZN) had a few days with missing values scattered throughout the past five years.

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

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

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

```
ecmnmle(Data);
```



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

Estimate the mean and covariance as computed by `ecmnmle`.

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

```
ECMMean = 12×1

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

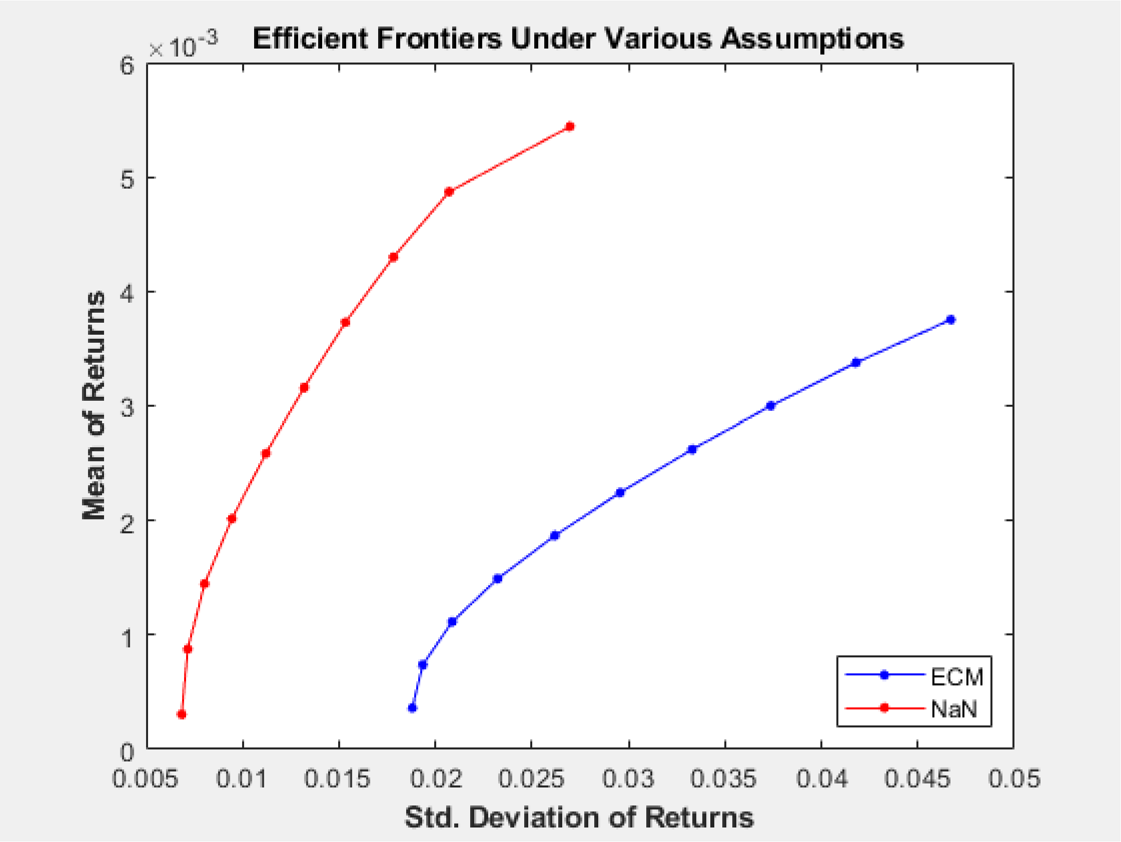
```
ECMCovar = 12×12

    0.0012    0.0005    0.0006    0.0005    0.0005    0.0003    0.0005    0.0003    0.0006    0.0
    0.0005    0.0024    0.0007    0.0006    0.0010    0.0004    0.0005    0.0003    0.0006    0.0
    0.0006    0.0007    0.0013    0.0007    0.0007    0.0003    0.0006    0.0004    0.0008    0.0
    0.0005    0.0006    0.0007    0.0009    0.0006    0.0002    0.0005    0.0003    0.0007    0.0
    0.0005    0.0010    0.0007    0.0006    0.0016    0.0006    0.0005    0.0003    0.0006    0.0
    0.0003    0.0004    0.0003    0.0002    0.0006    0.0022    0.0001    0.0002    0.0002    0.0
    0.0005    0.0005    0.0006    0.0005    0.0005    0.0001    0.0009    0.0003    0.0005    0.0
    0.0003    0.0003    0.0004    0.0003    0.0003    0.0002    0.0003    0.0005    0.0004    0.0
    0.0006    0.0006    0.0008    0.0007    0.0006    0.0002    0.0005    0.0004    0.0011    0.0
    0.0003    0.0004    0.0005    0.0004    0.0004    0.0001    0.0004    0.0003    0.0005    0.0
      ⋮
```

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);
```

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. Deviation of Returns');
ylabel('\bfMean of Returns');
hold off
```



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:

```
Assets
```

```
Assets = 1x12 cell
    {'AAPL'}    {'AMZN'}    {'CSCO'}    {'DELL'}    {'EBAY'}    {'GOOG'}    {'HPQ'}    {'IBM'}
```

```
ECMWts
```

```
ECMWts = 10×12

    0.0358    0.0011         0         0         0    0.0989    0.0535    0.4676         0    0.3
    0.0654    0.0110         0         0         0    0.1877    0.0179    0.3899         0    0.3
    0.0923    0.0194         0         0         0    0.2784         0    0.3025         0    0.3
    0.1165    0.0264         0         0         0    0.3712         0    0.2054         0    0.2
    0.1407    0.0334         0         0         0    0.4639         0    0.1083         0    0.2
    0.1648    0.0403         0         0         0    0.5566         0    0.0111         0    0.2
    0.1755    0.0457         0         0         0    0.6532         0         0         0    0.1
    0.1845    0.0509         0         0         0    0.7502         0         0         0    0.0
    0.1093    0.0174         0         0         0    0.8733         0         0         0
         0         0         0         0         0    1.0000         0         0         0
```

```
NaNWts
```

```
NaNWts = 10×12

         0         0         0    0.1185         0    0.0522    0.0824    0.1779         0    0.5
    0.0576         0         0    0.1219         0    0.0854    0.1274    0.0460         0    0.5
    0.1248         0         0    0.0952         0    0.1195    0.1674         0         0    0.4
    0.1969         0         0    0.0529         0    0.1551    0.2056         0         0    0.3
    0.2690         0         0    0.0105         0    0.1906    0.2438         0         0    0.2
    0.3414         0         0         0         0    0.2265    0.2782         0         0    0.0
    0.4235         0         0         0         0    0.2639    0.2788         0         0
    0.5245         0         0         0         0    0.3034    0.1721         0         0
    0.6269         0         0         0         0    0.3425    0.0306         0         0
    1.0000         0         0         0         0         0         0         0         0
```

The naïve portfolios in `NaNWts` tend to favor 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 AAPL and to recommend increased weights in GOOG relative to the naïve weights.

To evaluate the impact of the 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');
```

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. To view the differences:

```
Assets
```

```
Assets = 1x12 cell
    {'AAPL'}    {'AMZN'}    {'CSCO'}    {'DELL'}    {'EBAY'}    {'GOOG'}    {'HPQ'}    {'IBM'}
```

```
StdMeanH'
```

```
ans = 1×12

    0.0010    0.0014    0.0010    0.0009    0.0011    0.0021    0.0009    0.0006    0.0009    0.0
```

```
StdMeanF'
```

```
ans = 1×12

    0.0010    0.0014    0.0010    0.0009    0.0011    0.0013    0.0009    0.0006    0.0009    0.0
```

```
StdMeanH' - StdMeanF'
```

```
ans = 1×12
10^-3 ×

   -0.0000    0.0021   -0.0000   -0.0000   -0.0000    0.7742   -0.0000   -0.0000   -0.0000   -0.0
```

The two assets with missing data, AMZN and GOOG, are the only assets to have differences due to missing information.

See Also

`mvnrmle` | `mvnrstd` | `mvnrfish` | `mvnrobj` | `ecmmvnrmle` | `ecmmvnrstd` | `ecmmvnrfish` | `ecmmvnrobj` | `ecmlsrmle` | `ecmlsrobj` | `ecmmvnrstd` | `ecmmvnrfish` | `ecmnmle` | `ecmnstd` | `ecmnfish` | `ecmnhess` | `ecmnobj` | `convert2sur` | `ecmninit`

Related Examples

- “Multivariate Normal Regression” on page 9-2
- “Maximum Likelihood Estimation with Missing Data” on page 9-7
- “Valuation with Missing Data” on page 9-26

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.

The following example shows how to use the missing data regression functions to estimate the coefficients of the CAPM.

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.Beta is 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. 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 $k = 1, \ldots, m$ and assets $i = 1, \ldots, n$, where α_i is a parameter that specifies the nonsystematic return of an asset, β_i is the asset beta, and $V_{k,i}$ is the residual error for each asset with associated random variable V_i.

The collection of parameters $\alpha_1, \ldots, \alpha_n$ are called asset alphas. The strict form of the CAPM specifies that alphas must be zero and that deviations from zero are the result of temporary disequilibria. In practice, however, assets may have nonzero alphas, where much of active investment management is devoted to the search for assets with exploitable nonzero alphas.

To allow for the possibility of nonzero alphas, the estimation model generally seeks to estimate alphas and to perform tests to determine if the alphas are statistically equal to zero.

The residual errors V_i are assumed to have moments

$$E[V_i] = 0$$

and

$$E\left[V_i V_j\right] = S_{ij}$$

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

The square root of the residual variance of each asset, for example, sqrt(S_{ii}) for $i = 1, \ldots, n$, is said to be the residual or nonsystematic risk of the asset since it characterizes the residual variation in asset prices that 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     Attributes
  Assets         1x14              1568  cell
  Data        1471x14            164752  double
  Dates       1471x1              11768  double
```

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     Attributes

  Covar0      12x12            1152  double
  Param0      24x1              192  double
```

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

`mvnrmle` | `mvnrstd` | `mvnrfish` | `mvnrobj` | `ecmmvnrmle` | `ecmmvnrstd` | `ecmmvnrfish` | `ecmmvnrobj` | `ecmlsrmle` | `ecmlsrobj` | `ecmmvnrstd` | `ecmmvnrfish` | `ecmnmle` | `ecmnstd` | `ecmnfish` | `ecmnhess` | `ecmnobj` | `convert2sur` | `ecmninit`

Related Examples

- “Multivariate Normal Regression” on page 9-2

- “Maximum Likelihood Estimation with Missing Data” on page 9-7
- “Multivariate Normal Regression Types” on page 9-13
- “Portfolios with Missing Data” on page 9-21

Capital Asset Pricing Model with Missing Data

This example illustrates implementation of the Capital Asset Pricing Model (CAPM) in the presence of missing data.

The Capital Asset Pricing Model

The Capital Asset Pricing Model (CAPM) is a venerable but often-maligned tool to characterize comovements between asset and market prices. Although many issues arise in its implementation and interpretation, one problem that practitioners face is to estimate the coefficients of the CAPM with incomplete stock price data.

Given a host of assumptions that can be found in the references (see Sharpe [3 on page 9-38], Lintner [2 on page 9-38], Jarrow [1 on page 9-38], and Sharpe, et. al. [4 on page 9-38]), the CAPM concludes that asset returns have a linear relationship with market returns. Specifically, given the return of all stocks that constitute a market denoted as `M` and the return of a riskless asset denoted as `C`, the CAPM states that the return of each asset `R(i)` in the market has the expectational form

```
E[R(i)] = C + b(i) * (E[M] - C)
```

for assets `i = 1, ... , n`, where `b(i)` is a parameter that specifies the degree of comovement between a given asset and the underlying market. In words, the expected return of each asset is equal to the return on a riskless asset plus a risk-adjusted expected market return net of riskless asset returns. The collection of parameters `b(1), ... , b(n)` are called asset betas.

Note that the beta of an asset has the form

```
b(i) = cov(R(i),M)/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 equal to 1, the asset is said to move with the market; if an asset has a beta greater 1, the asset is said to be more volatile than the market; and if an asset has a beta less than 1, the asset is said to be less volatile than the market.

Estimation of the CAPM

The standard form of the CAPM model for estimation 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) = a(i) + C(k) + b(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 non-systematic return of an asset, `b(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 non-zero alphas, where much of active investment management is devoted to the search for assets with exploitable non-zero alphas.

To allow for the possibility of non-zero 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(i,j)
```

for assets i,j = 1, ... , n, where the parameters S(1,1), ... , S(n,n) are called residual or non-systematic variances/covariances.

The square root of the residual variance of each asset, i.e., sqrt(S(i,i)) for i = 1, ... , n, is said to be the residual or non-systematic risk of the asset since it characterizes the residual variation in asset prices that cannot be explained by variations in market prices.

Estimation with Missing Data

Although betas can be estimated for companies with sufficiently long histories of asset returns, it is extremely difficult to estimate betas for recent IPOs. However, if a collection of sufficiently-observable companies exists that can be expected to have some degree of correlation with the new company's stock price movements, for example, companies within the same industry as the new company, then it is possible to obtain imputed estimates for new company betas with the missing-data regression routines in the Financial Toolbox™.

Separate Estimation of Some Technology Stock Betas

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

First, load dates, total returns, and ticker symbols for the twelve stocks from the MAT-file CAPMuniverse.

```
load CAPMuniverse
whos Assets Data Dates
```

```
  Name            Size                Bytes  Class     Attributes

  Assets          1x14                 1568  cell                
  Data         1471x14               164752  double              
  Dates        1471x1                 11768  double              
```

```
Dates = datetime(Dates,'ConvertFrom','datenum');
```

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)
```

```
ans = 1x7 cell
    {'AAPL'}    {'AMZN'}    {'CSCO'}    {'DELL'}    {'EBAY'}    {'GOOG'}    {'HPQ'}
```

```
Assets(8:14)
```

```
ans = 1x7 cell
    {'IBM'}    {'INTC'}    {'MSFT'}    {'ORCL'}    {'YHOO'}    {'MARKET'}    {'CASH'}
```

The data covers the period from January 1, 2000 to November 7, 2005 with daily total returns. Two stocks in this universe have missing values that are represented by NaNs. One of the two stocks had an IPO during this period and, consequently, has significantly less data than the other stocks.

The first step is to 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);

Alpha = NaN(1, length(NumAssets));
Beta = NaN(1, length(NumAssets));
Sigma = NaN(1, length(NumAssets));
StdAlpha = NaN(1, length(NumAssets));
StdBeta = NaN(1, length(NumAssets));
StdSigma = NaN(1, length(NumAssets));
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 the CAPM for each asset separately.
    [Param, Covar] = ecmmvnrmle(TestData, TestDesign);

    % Estimate the ideal standard errors for covariance parameters.
    [StdParam, StdCovar] = ecmmvnrstd(TestData, TestDesign, Covar, 'fisher');

    % Estimate the sample standard errors for model parameters.
    StdParam = ecmmvnrstd(TestData, TestDesign, Covar, 'hessian');

    % Set up results for the output.
    Alpha(i) = Param(1);
    Beta(i) = Param(2);
    Sigma(i) = sqrt(Covar);

    StdAlpha(i) = StdParam(1);
    StdBeta(i) = StdParam(2);
    StdSigma(i) = sqrt(StdCovar);
end

displaySummary('Separate', StartDate, EndDate, NumAssets, Assets, Alpha, StdAlpha, Beta, StdBeta
```

```
Separate 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    0.0006 (  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 would seem, 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 non-systematic risks. Instead of *t*-statistics, the associated standard errors for the residual standard deviations are enclosed in parentheses.

Grouped Estimation of Some Technology Stock Betas

To estimate stock betas for all twelve stocks, set up a joint regression model that groups all twelve stocks within a single design (since each stock has the same design matrix, this model is actually an example of seemingly-unrelated regression). The function to estimate model parameters is ecmmvnrmle and the function to estimate standard errors is ecmmvnrstd.

Since GOOG has a significant number of missing values, a direct use of the missing data function 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 (CAPMgroupparam) and is used as initial estimates to compute stock betas.

```
load CAPMgroupparam
whos Param0 Covar0
```

```
  Name          Size            Bytes  Class     Attributes

  Covar0      12x12             1152  double              
  Param0      24x1               192  double              
```

Now estimate the parameters for the collection of twelve stocks.

```
NumParams = 2 * NumAssets;

% Set up the 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 the CAPM for all assets together with initial parameter estimates.
[Param, Covar] = ecmmvnrmle(TestData, TestDesign, [], [], [], Param0, Covar0);

% Estimate the ideal standard errors for covariance parameters.
[StdParam, StdCovar] = ecmmvnrstd(TestData, TestDesign, Covar, 'fisher');

% Estimate the sample standard errors for model parameters.
StdParam = ecmmvnrstd(TestData, TestDesign, Covar, 'hessian');

% Set up results for the 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));

displaySummary('Grouped', StartDate, EndDate, NumAssets, Assets, Alpha, StdAlpha, Beta, StdBeta,
```

```
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, notice that the beta estimates for AMZN and GOOG (which are the two stocks with missing values) are different from the estimates derived for each stock separately. Since AMZN has few missing values, the differences in the estimates are small. With GOOG, however, the differences are more pronounced.

The *t*-statistic for the beta estimate of GOOG is now significant at the 99.5% level of significance. Note, however, that the *t*-statistics for beta estimates are based on standard errors from the sample Hessian which, in contrast to the Fisher information matrix, accounts for the increased uncertainty in an estimate due to missing values. If the *t*-statistic is obtained from the more optimistic Fisher information matrix, the *t*-statistic for GOOG is `8.25`. Thus, despite the increase in uncertainty due to missing data, GOOG nonetheless has a statistically-significant estimate for beta.

Finally, note that the beta estimate for GOOG is `0.62` - a value that may require some explanation. Whereas the market has been volatile over this period with sideways price movements, GOOG has steadily appreciated in value. Consequently, it is less correlated than the market, which, in turn, implies that it is less volatile than the market with a beta less than 1.

References

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

[2] J. Lintner. "The Valuation of Risk Assets and the Selection of Risky Investments in Stocks." *Review of Economics and Statistics*. Vol. 14, 1965, pp. 13-37.

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

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

Utility Functions

```
function displaySummary(regressionType, StartDate, EndDate, NumAssets, Assets, Alpha, StdAlpha, B
fprintf(1,'%s regression with daily total return data from %s to %s ...\n', ...
    regressionType, string(StartDate),string(EndDate));
fprintf(1,'  %4s %-20s %-20s %-20s\n',' ','Alpha','Beta','Sigma');
fprintf(1,'  ---- -------------------- -------------------- --------------------\n');

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

end
```

Solving Sample Problems

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
```

```
Prices =

  117.7622
  101.1534
  763.3932
```

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
```

```
PortfolioAmounts =

  283.0562
  329.5324
   43.6647
```

Step 4

Calculate the modified duration and convexity of the portfolio. The portfolio duration or convexity is a weighted average of the durations or convexities of the individual bonds. Calculate the first- and second-order approximations of the percent price change as a function of the change in the level of interest rates.

```
PortfolioDuration  = PortfolioWeights' * Durations;
PortfolioConvexity = PortfolioWeights' * Convexities;
PercentApprox1 = -PortfolioDuration * dY * 100

PercentApprox2 =  PercentApprox1 + ...
PortfolioConvexity*dY^2*100/2.0
```

```
PercentApprox1 =

   -2.0636


PercentApprox2 =

   -2.0321
```

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
```

```
PriceApprox1 =

   9.7936e+04


PriceApprox2 =

   9.7968e+04
```

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)
```

```
NewPrice =

   9.7968e+04
```

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 10-6.
- 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 $97,968.90.
- The true new portfolio price (`NewPrice`) for this yield curve shift is $97,968.51.
- 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

`bnddury` | `bndconvy` | `bndprice` | `bndkrdur` | `blsprice` | `blsdelta` | `blsgamma` | `blsvega` | `zbtprice` | `zero2fwd` | `zero2disc`

Related Examples

- “Pricing and Analyzing Equity Derivatives” on page 2-39
- “Greek-Neutral Portfolios of European Stock Options” on page 10-14
- “Bond Portfolio for Hedging Duration and Convexity” on page 10-6
- “Bond Prices and Yield Curve Parallel Shifts” on page 10-9
- “Bond Prices and Yield Curve Nonparallel Shifts” on page 10-12
- “Term Structure Analysis and Interest-Rate Swaps” on page 10-18
- “Plotting Sensitivities of an Option” on page 10-25
- “Plotting Sensitivities of a Portfolio of Options” on page 10-27

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 10-2. 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 thing 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
```

```
Prices =

  530.4248
  994.4065
  273.4051
```

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

Weights =

   -0.3043
    0.7130
    0.5913
```

Step 4

Compute the duration and convexity of the hedge portfolio, which should now match the original portfolio.

```
PortfolioDuration  = Weights' * Durations
PortfolioConvexity = Weights' * Convexities

PortfolioDuration =

   10.3181

PortfolioConvexity =

  157.6346
```

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

HedgeAmounts =

  -57.3716
   71.7044
  216.2653
```

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

`bnddury` | `bndconvy` | `bndprice` | `bndkrdur` | `blsprice` | `blsdelta` | `blsgamma` | `blsvega` | `zbtprice` | `zero2fwd` | `zero2disc`

Related Examples

- “Pricing and Analyzing Equity Derivatives” on page 2-39
- “Greek-Neutral Portfolios of European Stock Options” on page 10-14
- “Sensitivity of Bond Prices to Interest Rates” on page 10-2
- “Bond Prices and Yield Curve Parallel Shifts” on page 10-9
- “Bond Prices and Yield Curve Nonparallel Shifts” on page 10-12
- “Term Structure Analysis and Interest-Rate Swaps” on page 10-18
- “Plotting Sensitivities of an Option” on page 10-25
- “Plotting Sensitivities of a Portfolio of Options” on page 10-27

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     = datetime(1995,1,15);
Maturity   = datetime( [2020, 4, 3;...
                         2025, 5,14;...
                         2019, 6, 9;...
                         2019, 2,25])
```

```
Maturity = 4x1 datetime
   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 = 4×1

  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 = datetime(year(Settle):2018, month(Settle), day(Settle)); % 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.

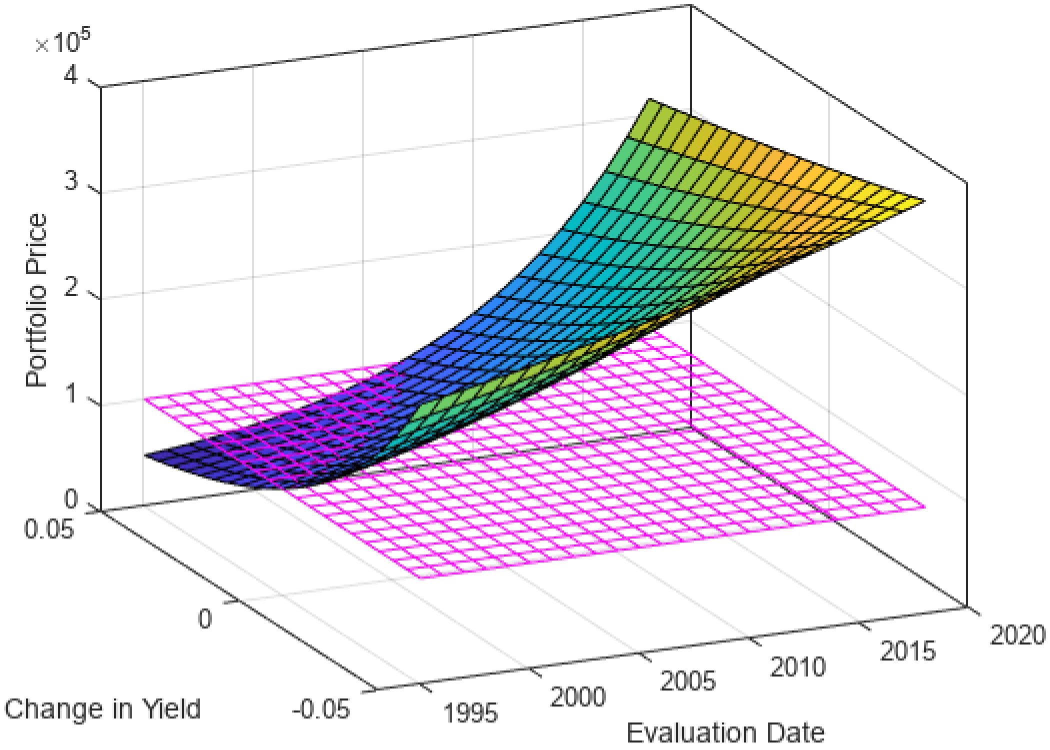
```
for i = 1:NumBonds

   [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');

xlim(datetime([1993,2020],1,1))
xlabel('Evaluation Date');
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

`bnddury` | `bndconvy` | `bndprice` | `bndkrdur` | `blsprice` | `blsdelta` | `blsgamma` | `blsvega` | `zbtprice` | `zero2fwd` | `zero2disc`

Related Examples

- “Pricing and Analyzing Equity Derivatives” on page 2-39
- “Greek-Neutral Portfolios of European Stock Options” on page 10-14
- “Sensitivity of Bond Prices to Interest Rates” on page 10-2
- “Bond Portfolio for Hedging Duration and Convexity” on page 10-6
- “Bond Prices and Yield Curve Nonparallel Shifts” on page 10-12
- “Term Structure Analysis and Interest-Rate Swaps” on page 10-18
- “Plotting Sensitivities of an Option” on page 10-25
- “Plotting Sensitivities of a Portfolio of Options” on page 10-27

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 = datetime(2008,12,2);
CouponRate = 5.500/100;
Maturity = datetime(2028,8,15);
Price = 128.68;
```

The interest-rate risk of this bond is hedged with the following four on-the-run Treasury bonds:

```
Maturity_30 = datetime(2038,5,15);  % 30-year bond
Coupon_30 = .045;
Price_30 = 124.69;

Maturity_10 = datetime(2018,11,15);  %10-year note
Coupon_10 = .0375;
Price_10 = 109.35;

Maturity_05 = datetime(2013,11,30);  % 5-year note
Coupon_05 = .02;
Price_05 = 101.67;

Maturity_02 = datetime(2010,11,30);  % 2-year note
Coupon_02 = .01250;
Price_02 =  100.72;
```

You can get the Treasury spot or zero curve from https://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(table(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(table(ZeroDates, ZeroRates), HedgeCoupon,...
Settle, HedgeMaturity, 'keyrates',[2 5 10 20])
```

```
HedgeKRD = 4×4

    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 = HedgeKRD.*[Price_30;Price_10;Price_05;Price_02]
```

```
HedgeDD = 4×4
10^3 ×

    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 = 1×4

    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

`bnddury` | `bndconvy` | `bndprice` | `bndkrdur` | `blsprice` | `blsdelta` | `blsgamma` | `blsvega` | `zbtprice` | `zero2fwd` | `zero2disc`

Related Examples

- “Pricing and Analyzing Equity Derivatives” on page 2-39
- “Greek-Neutral Portfolios of European Stock Options” on page 10-14
- “Sensitivity of Bond Prices to Interest Rates” on page 10-2
- “Bond Portfolio for Hedging Duration and Convexity” on page 10-6
- “Bond Prices and Yield Curve Parallel Shifts” on page 10-9
- “Term Structure Analysis and Interest-Rate Swaps” on page 10-18
- “Plotting Sensitivities of an Option” on page 10-25
- “Plotting Sensitivities of a Portfolio of Options” on page 10-27

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. For more information, see “Pricing and Analyzing Equity Derivatives” on page 2-39.

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)'
```

```
Gammas =

    0.0290    0.0353    0.0548    0.0074


Vegas =

   17.4293   20.0347    9.5837   83.5225
```

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)]
```

```
Prices =

    6.3441    6.6035    4.2993   22.7694


Deltas =

    0.5856   -0.6255    0.7003   -0.4830
```

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 1-11.

```
A = [Deltas; Gammas; Vegas; Prices];
b = [0; 0; 0; 17000];
OptionQuantities = A\b % Quantity (number) of each option.
OptionQuantities =

   1.0e+04 *

    2.2333
    0.6864
   -1.5655
   -0.4511
```

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
```

```
PortfolioValue =

       17000


PortfolioDelta =

   1.8190e-12


PortfolioGamma =

     0


PortfolioVega =

     0
```

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
```

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

bnddury | bndconvy | bndprice | bndkrdur | blsprice | blsdelta | blsgamma | blsvega | zbtprice | zero2fwd | zero2disc

Related Examples

- "Pricing and Analyzing Equity Derivatives" on page 2-39
- "Sensitivity of Bond Prices to Interest Rates" on page 10-2
- "Bond Portfolio for Hedging Duration and Convexity" on page 10-6
- "Bond Prices and Yield Curve Parallel Shifts" on page 10-9
- "Bond Prices and Yield Curve Nonparallel Shifts" on page 10-12
- "Term Structure Analysis and Interest-Rate Swaps" on page 10-18
- "Plotting Sensitivities of an Option" on page 10-25
- "Plotting Sensitivities of a Portfolio of Options" on page 10-27

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)
```

```
ZeroRates =

    0.0614
    0.0642
    0.0660
    0.0684
    0.0702
    0.0726
    0.0754
    0.0795
    0.0827
    0.0868
```

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)
```

```
ForwardRates =

    0.0614
    0.0670
    0.0695
    0.0758
    0.0774
    0.0846
    0.0925
    0.1077
    0.1089
    0.1239
```

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)
DiscountFactors =

    0.9704
    0.9387
    0.9073
    0.8739
    0.8416
    0.8072
    0.7718
    0.7320
    0.6945
    0.6537
```

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)
```

```
PresentValue =

    0.3460
```

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)
```

```
SwapFixedRate =

    0.0845
```

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

`bnddury` | `bndconvy` | `bndprice` | `bndkrdur` | `blsprice` | `blsdelta` | `blsgamma` | `blsvega` | `zbtprice` | `zero2fwd` | `zero2disc`

Related Examples

- “Pricing and Analyzing Equity Derivatives” on page 2-39
- “Greek-Neutral Portfolios of European Stock Options” on page 10-14
- “Sensitivity of Bond Prices to Interest Rates” on page 10-2
- “Bond Portfolio for Hedging Duration and Convexity” on page 10-6
- “Bond Prices and Yield Curve Parallel Shifts” on page 10-9
- “Bond Prices and Yield Curve Nonparallel Shifts” on page 10-12
- “Plotting Sensitivities of an Option” on page 10-25
- “Plotting Sensitivities of a Portfolio of Options” on page 10-27

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 function `corr2cov`.

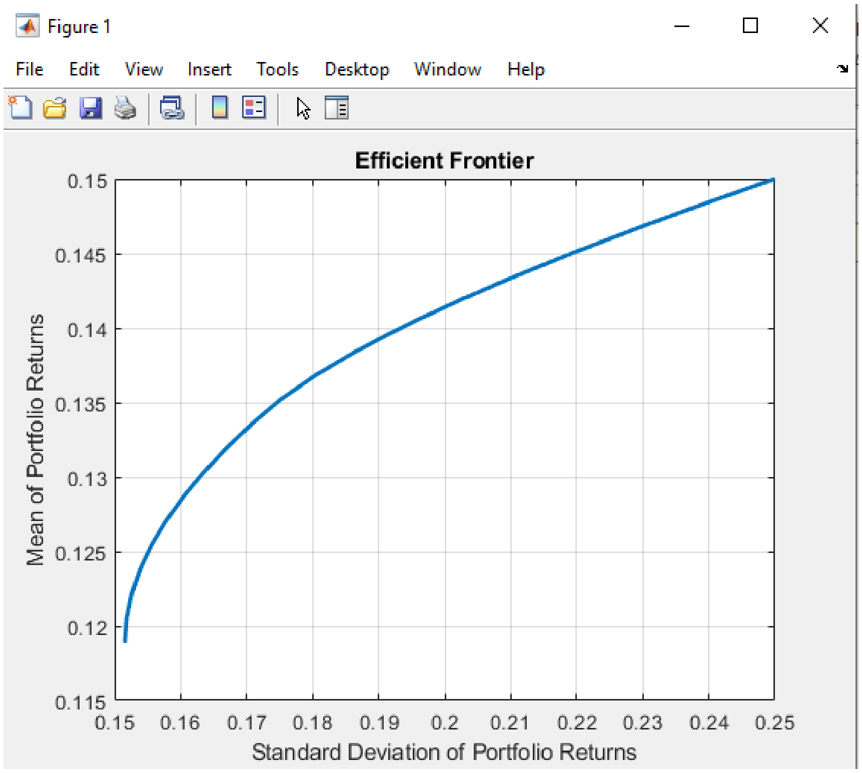
```
Covariances = corr2cov(STDs, Correlations)
```

```
Covariances =

    0.0400    0.0150    0.0144
    0.0150    0.0625    0.0135
    0.0144    0.0135    0.0324
```

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` object 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 4-17. For more information on migrating `portopt` code to `Portfolio`, see “portopt Migration to Portfolio Object” on page 3-11.

```
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.

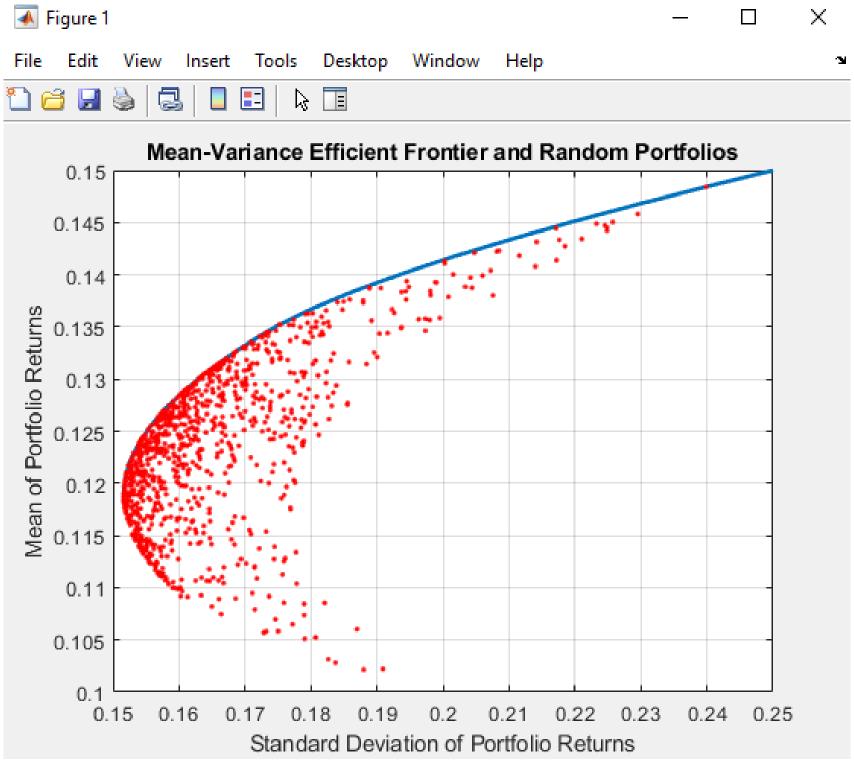
```
Total = sum(Weights, 2);     % Add the weights
Total = Total(:,ones(3,1)); % Make size-compatible matrix
Weights = Weights./Total;    % Normalize so sum = 1
```

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

`bnddury` | `bndconvy` | `bndprice` | `bndkrdur` | `blsprice` | `blsdelta` | `blsgamma` | `blsvega` | `zbtprice` | `zero2fwd` | `zero2disc` | `corr2cov` | `portopt`

Related Examples

- “Plotting Sensitivities of an Option” on page 10-25
- “Plotting Sensitivities of a Portfolio of Options” on page 10-27
- “Pricing and Analyzing Equity Derivatives” on page 2-39
- “Greek-Neutral Portfolios of European Stock Options” on page 10-14
- “Sensitivity of Bond Prices to Interest Rates” on page 10-2
- “Bond Portfolio for Hedging Duration and Convexity” on page 10-6
- “Bond Prices and Yield Curve Parallel Shifts” on page 10-9
- “Bond Prices and Yield Curve Nonparallel Shifts” on page 10-12
- “Term Structure Analysis and Interest-Rate Swaps” on page 10-18

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.

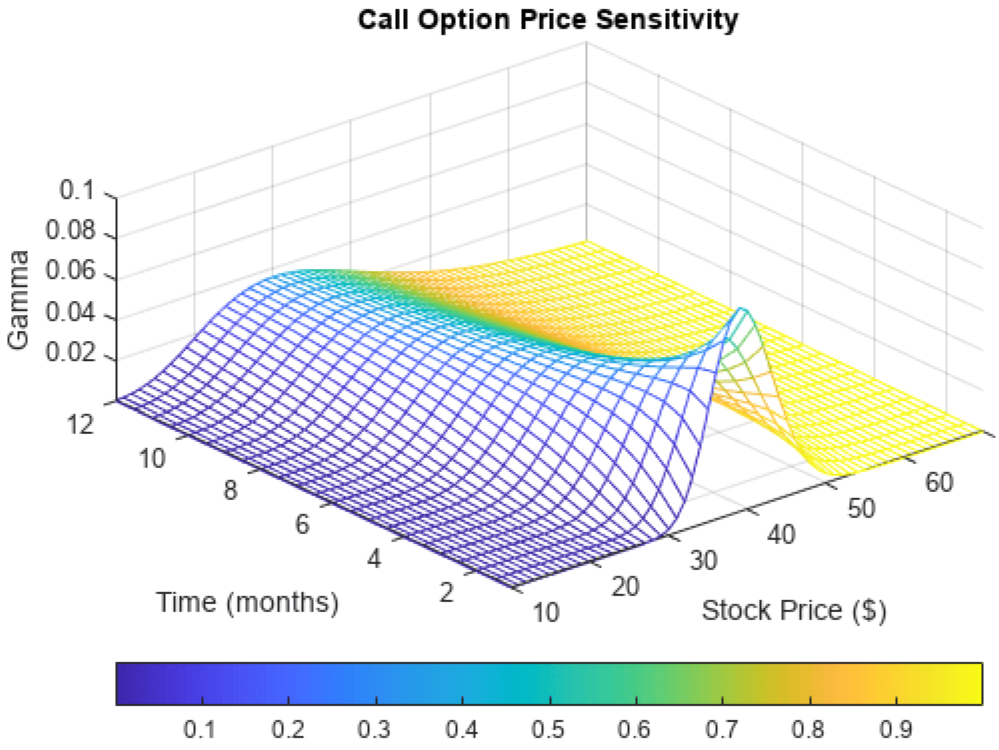
```
JSpan = ones(length(j),1);
NewRange = Range(JSpan,:);
Pad = ones(size(Newj));
```

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

`bnddury` | `bndconvy` | `bndprice` | `bndkrdur` | `blsprice` | `blsdelta` | `blsgamma` | `blsvega` | `zbtprice` | `zero2fwd` | `zero2disc` | `corr2cov` | `portopt`

Related Examples

- "Plotting Sensitivities of a Portfolio of Options" on page 10-27
- "Pricing and Analyzing Equity Derivatives" on page 2-39
- "Greek-Neutral Portfolios of European Stock Options" on page 10-14
- "Sensitivity of Bond Prices to Interest Rates" on page 10-2
- "Bond Portfolio for Hedging Duration and Convexity" on page 10-6
- "Bond Prices and Yield Curve Parallel Shifts" on page 10-9
- "Bond Prices and Yield Curve Nonparallel Shifts" on page 10-12
- "Term Structure Analysis and Interest-Rate Swaps" on page 10-18

Plotting Sensitivities of a Portfolio of Options

This example plots *gamma* as a function of price and time for a portfolio of ten 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.

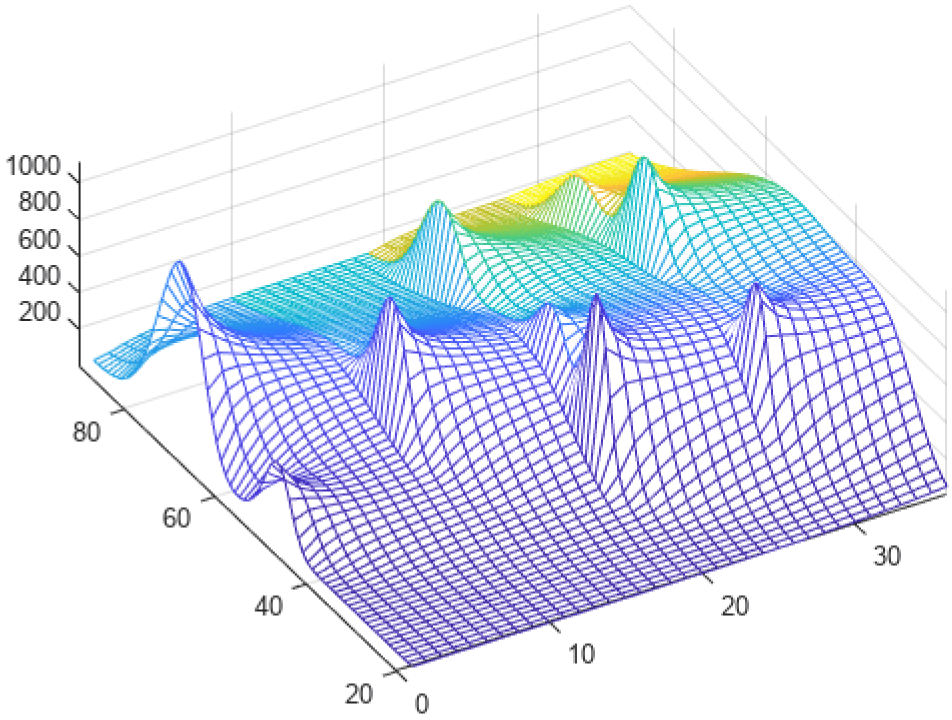
```
  T = (1:Time(i))';
  NewT = T(:,ones(PLen,1));
```

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
```

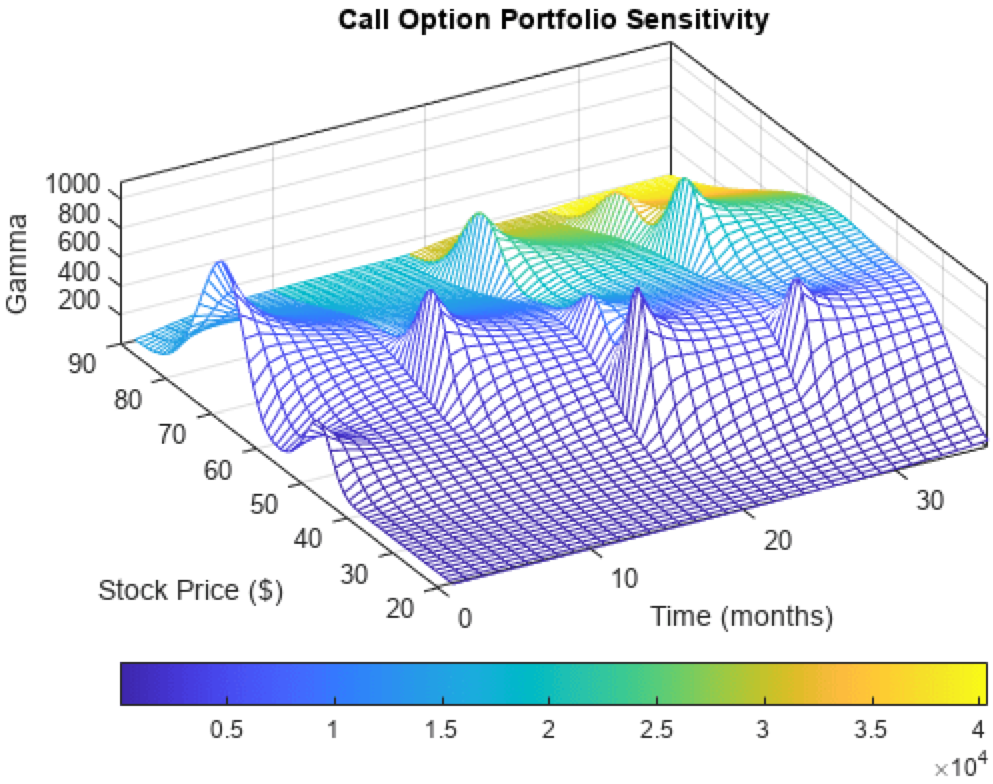
Draw the surface as a mesh, set the viewpoint, and reverse the *x*-axis because of the viewpoint. The axes range from `20` to `90`, `0` to `36`, and -∞ to ∞.

```
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

`bnddury` | `bndconvy` | `bndprice` | `bndkrdur` | `blsprice` | `blsdelta` | `blsgamma` | `blsvega` | `zbtprice` | `zero2fwd` | `zero2disc` | `corr2cov` | `portopt`

Related Examples

- “Plotting Sensitivities of an Option” on page 10-25
- “Pricing and Analyzing Equity Derivatives” on page 2-39
- “Greek-Neutral Portfolios of European Stock Options” on page 10-14
- “Sensitivity of Bond Prices to Interest Rates” on page 10-2
- “Bond Portfolio for Hedging Duration and Convexity” on page 10-6
- “Bond Prices and Yield Curve Parallel Shifts” on page 10-9
- “Bond Prices and Yield Curve Nonparallel Shifts” on page 10-12
- “Term Structure Analysis and Interest-Rate Swaps” on page 10-18

Bond Portfolio Optimization Using Portfolio Object

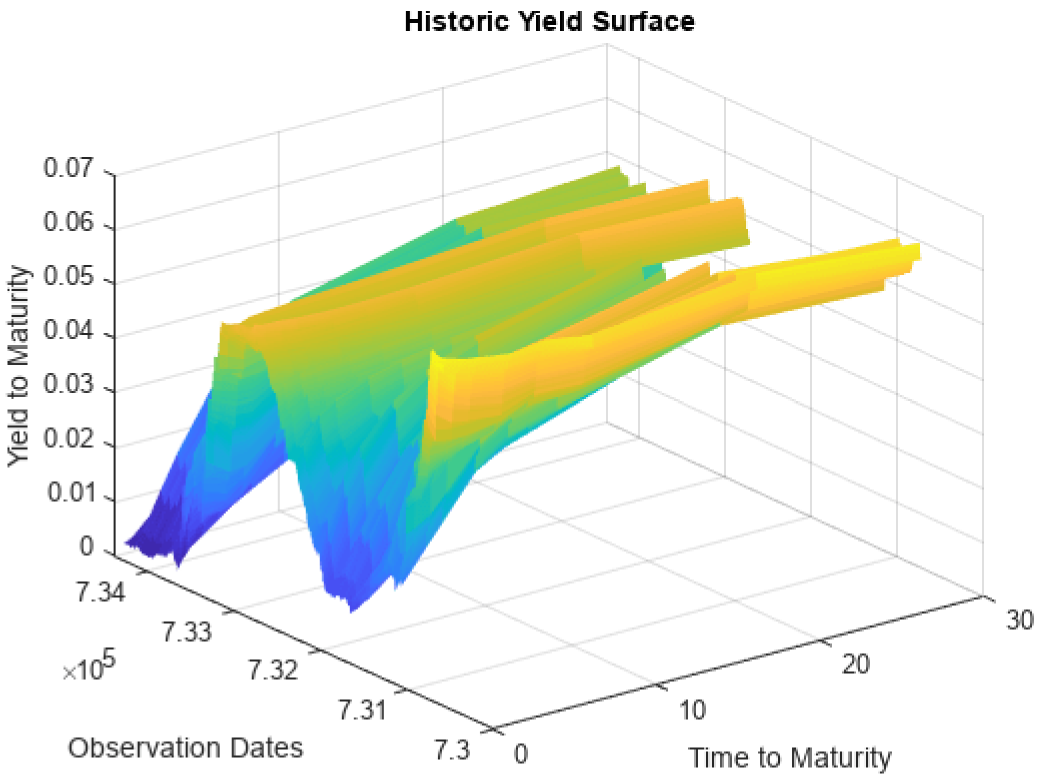
This example shows how to use a `Portfolio` object to construct an optimal portfolio of 10, 20, and 30 year treasuries that will be held for a period of one month. The workflow for the overall asset allocation process is:

1. Load market data — Historic daily treasury yields downloaded from FRED® are loaded.
2. Calculate market invariants — Daily changes in yield to maturity are chosen as invariants and assumed to be multivariate normal. Due to missing data for the 30 year bonds, an expectation maximization algorithm is used to estimate the mean and covariance of the invariants. The invariant's statistics are projected to the investment horizon.
3. Simulate invariants at horizon — Due to the high correlation and inherent structure in the yield curves, a principal component analysis is applied to the invariant statistics. Multivariate normal random draws are done in the PCA space. The simulations are transformed back into the invariant space using the PCA loadings.
4. Calculate distribution of returns at horizon — The simulated monthly changes in the yield curve are used to calculate the yield for the portfolio securities at the horizon. This requires interpolating values off of the simulated yield curves since the portfolio securities will have maturities that are one month less than 10, 20 and 30 years. Profit and loss for each scenario/security is calculated by pricing the treasuries using the simulated and interpolated yields. Simulated linear returns and their statistics are calculated from the prices.
5. Optimize asset allocation — Using a `Portfolio` object, mean-variance optimization is performed on the treasury returns statistics to calculate optimal portfolio weights for ten points along the efficient frontier. The investor preference is to choose the portfolio that is closest to the mean value of possible Sharpe ratios.

Load Market Data

Load the historic yield-to-maturity data for the series: DGS6MO, DGS1, DGS2, DGS3, DGS5, DGS7, DGS10, DGS20, DGS30 for the dates: Sep 1, 2000 to Sep 1, 2010 obtained from: https://fred.stlouisfed.org/categories/115 Note: Data is downloaded using Datafeed Toolbox™ using commands like: `>> conn = fred; >> data = fetch(conn,'DGS10','9/1/2000','9/1/2010');` results have been aggregated and stored in a binary `HistoricalYTMData.mat` file for convinience.

```
histData = load('HistoricalYTMData.mat');
% Time to maturity for each series
tsYTMMats = histData.tsYTMMats;
% Dates that rates were observed
tsYTMObsDates = histData.tsYTMObsDates;
% Observed rates
tsYTMRates = histData.tsYTMRates;

% Visualize the yield surface
[X,Y] = meshgrid(tsYTMMats,tsYTMObsDates);
surf(X,Y,tsYTMRates,EdgeColor='none')
xlabel('Time to Maturity')
ylabel('Observation Dates')
zlabel('Yield to Maturity')
title('Historic Yield Surface')
```



Calculate Market Invariants

For market invariants, use the standard: daily changes in yield to maturity for each series. You can estimate their statistical distribution to be multivariate normal. IID analysis on each invariant series produces decent results - more so in the "independent" factor than "identical". A more thorough modeling using more complex distributions and/or time series models is beyond the scope of this example. What will need to be accounted for is the estimation of distribution parameters in the the presence of missing data. The 30 year bonds were discontinued for a period between Feb 2002 and Feb 2006, so there are no yields for this time period.

```
% Invariants are assumed to be daily changes in YTM rates.
tsYTMRateDeltas = diff(tsYTMRates);
```

About 1/3 of the 30 year rates (column 9) are missing from the original data set. Rather than throw out all these observations, an expectation maximization routine `ecmnmle` is used to estimate the mean and covariance of the invariants. The default option (NaN skip for initial estimates) is used.

```
[tsInvMu,tsInvCov] = ecmnmle(tsYTMRateDeltas);
```

Calculate standard deviations and correlations using `cov2corr`.

```
[tsInvStd,tsInvCorr] = cov2corr(tsInvCov);
```

The investment horizon is 1 month. (21 business days between 9/1/2010 and 10/1/2010). Since the invariants are summable and the means and variances of normal distributions are normal, you can project the invariants to the investment horizon as follows:

```
hrznInvMu = 21*tsInvMu';
hrznInvCov = 21*tsInvCov;
[hrznInvStd,hrznInvCor] = cov2corr(hrznInvCov);
```

The market invariants projected to the horizon have the following statistics:

```
disp('Mean:');
```

```
Mean:
```

```
disp(hrznInvMu);
```

```
   1.0e-03 *

   -0.5149   -0.4981   -0.4696   -0.4418   -0.3788   -0.3268   -0.2604   -0.2184   -0.1603
```

```
disp('Standard Deviation:');
```

```
Standard Deviation:
```

```
disp(hrznInvStd);
```

```
    0.0023    0.0024    0.0030    0.0032    0.0033    0.0032    0.0030    0.0027    0.0026
```

```
disp('Correlation:');
```

```
Correlation:
```

```
disp(hrznInvCor);
```

```
    1.0000    0.8553    0.5952    0.5629    0.4980    0.4467    0.4028    0.3338    0.3088
    0.8553    1.0000    0.8282    0.7901    0.7246    0.6685    0.6175    0.5349    0.4973
    0.5952    0.8282    1.0000    0.9653    0.9114    0.8589    0.8055    0.7102    0.6642
    0.5629    0.7901    0.9653    1.0000    0.9519    0.9106    0.8664    0.7789    0.7361
    0.4980    0.7246    0.9114    0.9519    1.0000    0.9725    0.9438    0.8728    0.8322
    0.4467    0.6685    0.8589    0.9106    0.9725    1.0000    0.9730    0.9218    0.8863
    0.4028    0.6175    0.8055    0.8664    0.9438    0.9730    1.0000    0.9562    0.9267
    0.3338    0.5349    0.7102    0.7789    0.8728    0.9218    0.9562    1.0000    0.9758
    0.3088    0.4973    0.6642    0.7361    0.8322    0.8863    0.9267    0.9758    1.0000
```

Simulate Market Invariants at Horizon

The high correlation is not ideal for simulation of the distribution of invariants at the horizon (and ultimately security prices). Use a principal component decomposition to extract orthogonal invariants. This could also be used for dimension reduction, however since the number of invariants is still relatively small, retain all nine components for more accurate reconstruction. However, missing values in the market data prevents you from estimating directly off of the time series data. Instead, this can be done directly off of the covariance matrix

```
% Perform PCA decomposition using the invariants' covariance.
[pcaFactorCoeff,pcaFactorVar,pcaFactorExp] = pcacov(hrznInvCov);

% Keep all components of pca decompositon.
numFactors = 9;

% Create a PCA factor covariance matrix.
pcaFactorCov = corr2cov(sqrt(pcaFactorVar),eye(numFactors));

% Define the number of simulations (random draws).
numSim = 10000;
```

```
% Fix the random seed for reproducible results.
stream = RandStream('mrg32k3a');
RandStream.setGlobalStream(stream);

% Take random draws from a multivariate normal distribution with zero mean
% and diagonal covariance.
pcaFactorSims = mvnrnd(zeros(numFactors,1),pcaFactorCov,numSim);

% Transform to horizon invariants and calculate the statistics.
hrznInvSims = pcaFactorSims*pcaFactorCoeff + repmat(hrznInvMu,numSim,1);
hrznInvSimsMu = mean(hrznInvSims);
hrznInvSimsCov = cov(hrznInvSims);
[hrznInvSimsStd,hrznInvSimsCor] = cov2corr(hrznInvSimsCov);
```

The simulated invariants have very similar statistics to the original invariants:

```
disp('Mean:');
```

```
Mean:
```

```
disp(hrznInvSimsMu);
```

```
   1.0e-03 *

   -0.5222   -0.5118   -0.4964   -0.4132   -0.3255   -0.3365   -0.2508   -0.2171   -0.1636
```

```
disp('Standard Deviation:');
```

```
Standard Deviation:
```

```
disp(hrznInvSimsStd);
```

```
    0.0016    0.0047    0.0046    0.0025    0.0040    0.0017    0.0007    0.0005    0.0004
```

```
disp('Correlation:');
```

```
Correlation:
```

```
disp(hrznInvSimsCor);
```

```
    1.0000    0.8903    0.7458   -0.4746   -0.4971    0.4885   -0.2353   -0.0971    0.1523
    0.8903    1.0000    0.9463   -0.7155   -0.6787    0.5164   -0.2238   -0.0889    0.2198
    0.7458    0.9463    1.0000   -0.8578   -0.7610    0.4659   -0.1890   -0.0824    0.2631
   -0.4746   -0.7155   -0.8578    1.0000    0.9093   -0.4999    0.2378    0.1084   -0.2972
   -0.4971   -0.6787   -0.7610    0.9093    1.0000   -0.7159    0.3061    0.1118   -0.2976
    0.4885    0.5164    0.4659   -0.4999   -0.7159    1.0000   -0.5360   -0.1542    0.1327
   -0.2353   -0.2238   -0.1890    0.2378    0.3061   -0.5360    1.0000    0.3176   -0.1108
   -0.0971   -0.0889   -0.0824    0.1084    0.1118   -0.1542    0.3176    1.0000    0.0093
    0.1523    0.2198    0.2631   -0.2972   -0.2976    0.1327   -0.1108    0.0093    1.0000
```

Calculate Distribution of Security Returns at Horizon

The portfolio will consist of 10, 20, and 30 year maturity treasuries. For simplicity, assume that these are new issues on the settlement date and are priced at market value inferred from the current yield curve. Profit and loss distributions are calculated by pricing each security along each simulated yield at the horizon and subtracting the purchase price. The horizon prices require nonstandard time to maturity yields. These are calculated using cubic spline interpolation. Simulated linear returns are their statistics that are calculated from the profit and loss scenarios.

```
% Define the purchase and investment horizon dates.
settleDate = datetime(2010,9,1);
hrznDate = datetime(2010,10,1);

% Define the maturity dates for new issue treasuries purchased on the
% settle date.
treasuryMaturities = [datetime(2020,9,1) , datetime(2030,9,1) , datetime(2040,9,1)];

% Select the observed yields for the securities of interest on the
% settle date.
treasuryYTMAtSettle = tsYTMRates(end,7:9);

% Initialize arrays for later use.
treasuryYTMAtHorizonSim = zeros(numSim,3);
treasuryPricesAtSettle = zeros(1,3);
treasuryPricesAtHorizonSim = zeros(numSim,3);

% Use actual/actual day count basis with annualized yields.
basis = 8;
```

Price the treasuries at settle date with `bndprice` using the known yield to maturity. For simplicity, assume that none of these securities include coupon payments. Although the prices may not be accurate, the overall structure/relationships between values is preserved for the asset allocation process.

```
for j=1:3
    treasuryPricesAtSettle(j) = bndprice(treasuryYTMAtSettle(j),0,settleDate,...
                                         treasuryMaturities(j),'basis',basis);
end
```

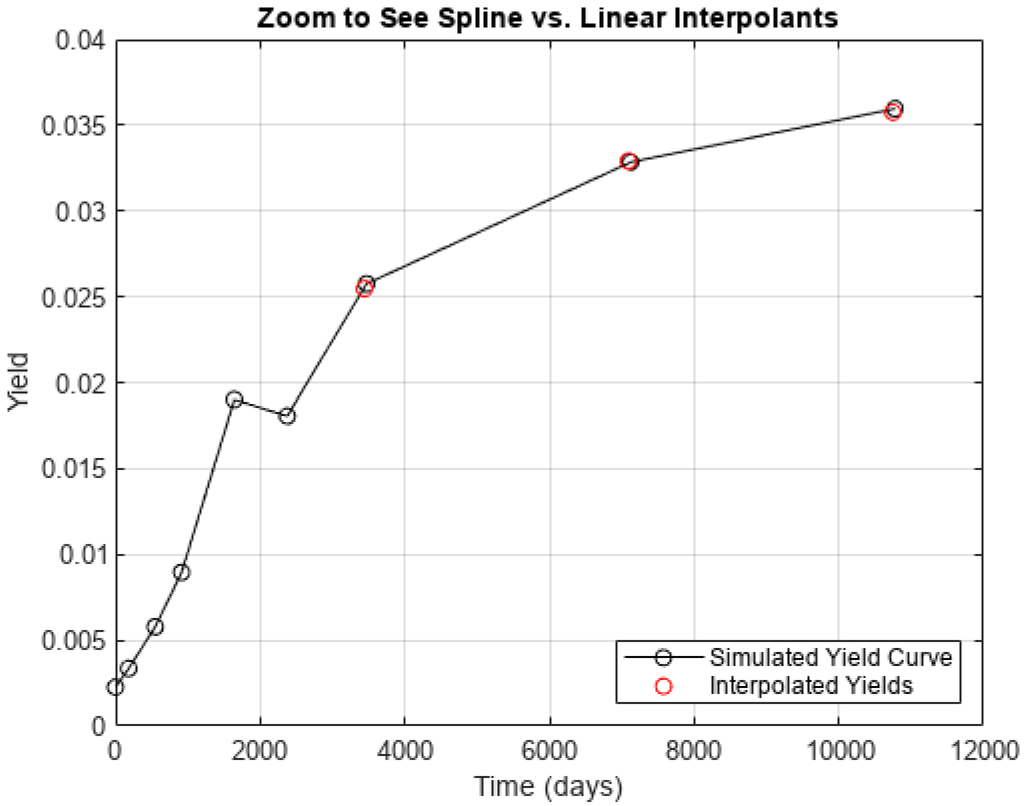
To price the treasuries at the horizon, you need to know yield to maturity at 9 years 11 months, 19 years 11 months, and 29 years 11 months for each simulation. You can approximate these using cubic spline interpolation using `interp1`.

```
% Transform the simulated invariants to YTM at the horizon.
hrznYTMRatesSims = repmat(tsYTMRates(end,:),numSim,1) + hrznInvSims;

hrznYTMMaturities = [datetime(2011,4,1),datetime(2011,10,1),datetime(2012,10,1),datetime(2013,10

% Convert the dates to numeric serial dates.
x = datenum(hrznYTMMaturities);
xi = datenum(treasuryMaturities);

% For numerical accuracy, shift the x values to start at zero.
minDate = min(x);
x = x - minDate;
xi = xi - minDate;
```

For each simulation and maturity approximate yield near 10,20, and 30 year nodes. Note that the effects of a spline fit vs. linear fit have a significant effect on the resulting ideal allocation. This is due to significant under-estimation of yield when using a linear fit for points just short of the known nodes.

```
for i=1:numSim
    treasuryYTMAtHorizonSim(i,:) = interp1(x,hrznYTMRatesSims(i,:),xi,'spline');
end

% Visualize a simulated yield curve with interpolation.
```

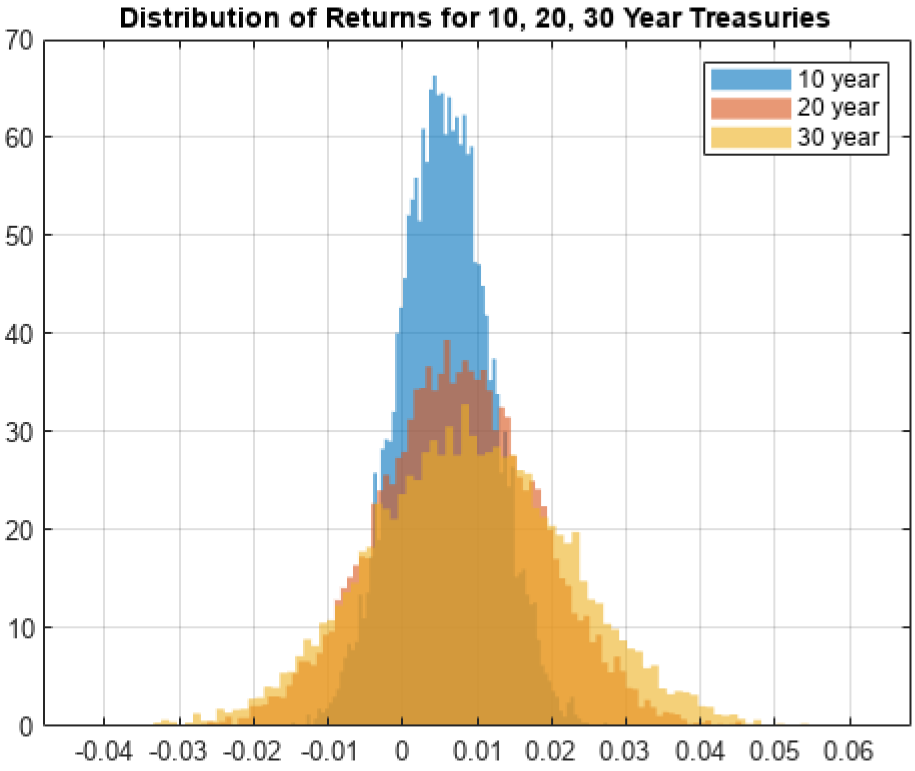
```
figure;
plot(x,hrznYTMRatesSims(1,:),'k-o',xi,treasuryYTMAtHorizonSim(1,:),'ro');
xlabel('Time (days)');
ylabel('Yield');
legend({'Simulated Yield Curve','Interpolated Yields'},'location','se');
grid on;
title('Zoom to See Spline vs. Linear Interpolants');
```



Price the treasuries at the horizon for each simulated yield to maturity. Note that the same assumptions are being made here as in the previous call to `bndprice`.

```
basis = 8*ones(numSim,1);
for j=1:3
    treasuryPricesAtHorizonSim(:,j) = bndprice(treasuryYTMAtHorizonSim(:,j),0,...
                                           hrznDate,treasuryMaturities(j),'basis',basis);
end

% Calculate the distribution of linear returns.
treasuryReturns = ( treasuryPricesAtHorizonSim - repmat(treasuryPricesAtSettle,numSim,1) )./repma

% Calculate the returns statistics.
retsMean = mean(treasuryReturns);
retsCov  = cov(treasuryReturns);
[retsStd,retsCor] = cov2corr(retsCov);

% Visualize the results for the 30 year treasury.
figure
histogram(treasuryReturns(:,1),100,Normalization='pdf',EdgeColor='none')
```

```
hold on
histogram(treasuryReturns(:,2),100,Normalization='pdf',EdgeColor='none')
histogram(treasuryReturns(:,3),100,Normalization='pdf',EdgeColor='none')
hold off
title('Distribution of Returns for 10, 20, 30 Year Treasuries');
grid on
legend({'10 year','20 year','30 year'});
```



Optimize Asset Allocation Using Portfolio Object

Asset allocation is optimized using a `Portfolio` object. Ten optimal portfolios (`NumPorts`) are calculated using `estimateFrontier` and their Sharpe ratios are calculated. The optimal portfolio, based on investor preference, is chosen to be the one that is closest to the maximum value of the Sharpe ratio.

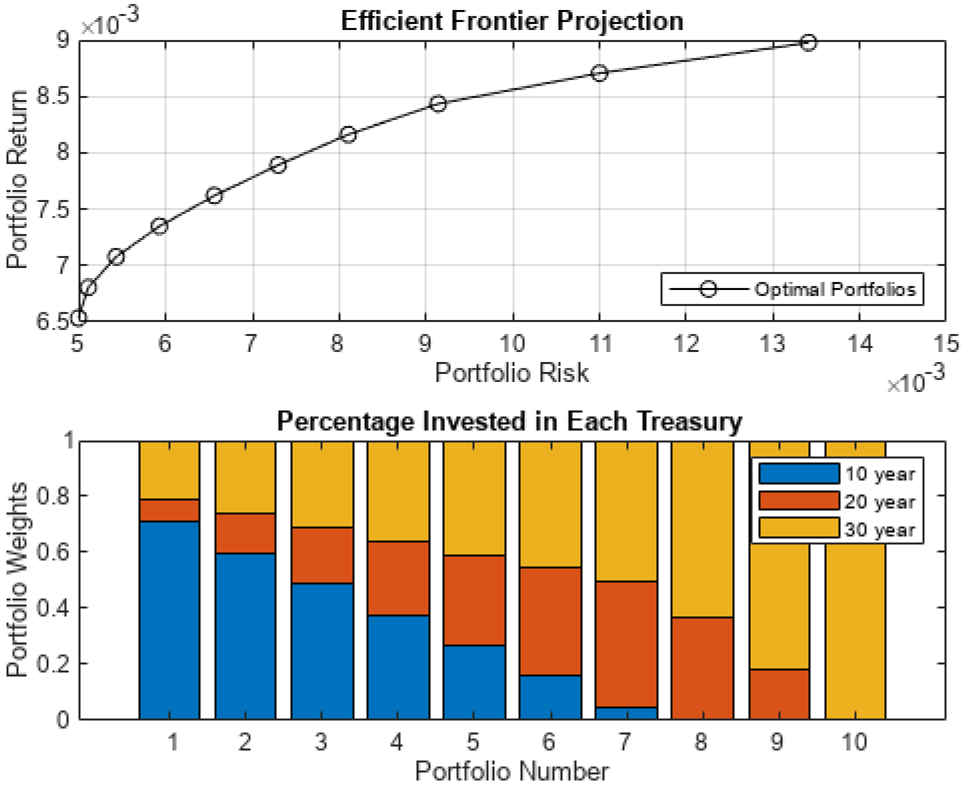
Create a `Portfolio` object using `Portfolio`. You can use `setAssetMoments` to set moments (mean and covariance) of asset returns for the `Portfolio` object and `setDefaultConstraints` to set up the portfolio constraints with nonnegative weights that sum to 1.

```
% Create a portfolio object.
p = Portfolio;
p = setAssetMoments(p,retsMean,retsCov);
p = setDefaultConstraints(p)
```

```
p = 
  Portfolio with properties:
          BuyCost: []
         SellCost: []
     RiskFreeRate: []
        AssetMean: [3×1 double]
       AssetCovar: [3×3 double]
    TrackingError: []
     TrackingPort: []
         Turnover: []
      BuyTurnover: []
     SellTurnover: []
             Name: []
        NumAssets: 3
        AssetList: []
         InitPort: []
      AInequality: []
      bInequality: []
        AEquality: []
        bEquality: []
       LowerBound: [3×1 double]
       UpperBound: []
      LowerBudget: 1
      UpperBudget: 1
      GroupMatrix: []
       LowerGroup: []
       UpperGroup: []
           GroupA: []
           GroupB: []
       LowerRatio: []
       UpperRatio: []
     MinNumAssets: []
     MaxNumAssets: []
        BoundType: [3×1 categorical]
```

Calculate ten points along the projection of the efficient frontier using `estimateFrontier` and `estimatePortMoments` to estimate moments of portfolio returns for the `Portfolio` object.

```
NumPorts = 10;
PortWts = estimateFrontier(p,NumPorts);
[PortRisk, PortReturn] = estimatePortMoments(p,PortWts);

% Visualize the portfolio.
figure;
subplot(2,1,1)
plot(PortRisk,PortReturn,'k-o');
xlabel('Portfolio Risk');
ylabel('Portfolio Return');
title('Efficient Frontier Projection');
legend('Optimal Portfolios','location','se');
grid on;

subplot(2,1,2)
bar(PortWts','stacked');
xlabel('Portfolio Number');
ylabel('Portfolio Weights');
title('Percentage Invested in Each Treasury');
legend({'10 year','20 year','30 year'});
```



Use `estimateMaxSharpeRatio` to estimate efficient portfolio that maximizes Sharpe ratio for the `Portfolio` object.

```
investorPortfolioWts = estimateMaxSharpeRatio(p);
```

The investor percentage allocation in 10,20, and 30 year treasuries is:

```
disp(investorPortfolioWts);
```

```
    0.6078
    0.1374
    0.2548
```

See Also

`tbilldisc2yield` | `tbillprice` | `tbillrepo` | `tbillval01` | `tbillyield` | `tbillyield2disc` | `Portfolio` | `setDefaultConstraints`

Related Examples

- “Pricing and Computing Yields for Fixed-Income Securities” on page 2-15
- “Computing Treasury Bill Price and Yield” on page 2-26
- “Term Structure of Interest Rates” on page 2-29
- “Sensitivity of Bond Prices to Interest Rates” on page 10-2
- “Bond Portfolio for Hedging Duration and Convexity” on page 10-6

- “Term Structure Analysis and Interest-Rate Swaps” on page 10-18
- “Creating the Portfolio Object” on page 4-24
- “Common Operations on the Portfolio Object” on page 4-32
- “Setting Up an Initial or Current Portfolio” on page 4-36
- “Asset Allocation Case Study” on page 4-172
- “Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152

More About

- “Treasury Bills Defined” on page 2-25

Hedge Options Using Reinforcement Learning Toolbox™

Outperform the traditional BSM approach using an optimal option hedging policy.

Option Modeling Using Black-Scholes-Merton Model

The Black-Scholes-Merton (BSM) model, which earned its creators a Nobel Prize in Economics in 1997, provides a modeling framework for pricing and analyzing financial derivatives or options. Options are financial instruments that derive their value from a particular underlying asset. The concept of *dynamic hedging* is fundamental to the BSM model. Dynamic hedging is the idea that, by continuously buying and selling shares in the relevant underlying asset, you can hedge the risk of the derivative instrument such that the risk is zero. This "risk-neutral" pricing framework is used to derive pricing formulae for many different financial instruments.

The simplest financial derivative is a European call option, which provides the buyer with the right, but not the obligation, to buy the underlying asset at a previously specified value (strike price) at a previously specified time (maturity).

You can use a BSM model to price a European call option. The BSM model makes the following simplifying assumptions:

- The behavior of the underlying asset is defined by geometric Brownian motion (GBM).
- There are no transaction costs.
- Volatility is constant.

The BSM dynamic hedging strategy is also called "delta-hedging," after the quantity *Delta*, which is the sensitivity of the option with respect to the underlying asset. In an environment that meets the previously stated BSM assumptions, using a delta-hedging strategy is an optimal approach to hedging an option. However, it is well-known that in an environment with transaction costs, the use of the BSM model leads to an inefficient hedging strategy. The goal of this example is to use Reinforcement Learning Toolbox™ to learn a strategy that outperforms the BSM hedging strategy, in the presence of transaction costs.

The goal of reinforcement learning (RL) is to train an agent to complete a task within an unknown environment. The agent receives observations and a reward from the environment and sends actions to the environment. The reward is a measure of how successful an action is with respect to completing the task goal.

The agent contains two components: a policy and a learning algorithm.

- The policy is a mapping that selects actions based on the observations from the environment. Typically, the policy is a function approximator with tunable parameters, such as a deep neural network.
- The learning algorithm continuously updates the policy parameters based on the actions, observations, and reward. The goal of the learning algorithm is to find an optimal policy that maximizes the cumulative reward received during the task.

In other words, reinforcement learning involves an agent learning the optimal behavior through repeated trial-and-error interactions with the environment without human involvement. For more information on reinforcement learning, see “What Is Reinforcement Learning?” (Reinforcement Learning Toolbox).

Cao [2 on page 10-47] describes the setup for reinforcement learning as:

- S_i is the state at time i.
- A_i is the action taken at i.
- R_{i+1} is the resulting reward at time $i + 1$.

The aim of reinforcement learning is to maximize expected future rewards. In this financial application of reinforcement learning, maximizing expected rewards is learning a delta-hedging strategy as an optimal approach to hedging a European call option.

This example follows the framework outlined in Cao [2 on page 10-47]. Specifically, an accounting profit and loss (P&L) formulation from that paper is used to set up the reinforcement learning problem and a deep deterministic policy gradient (DDPG) agent is used. This example does not exactly reproduce the approach from [2 on page 10-47] because Cao *et. al*. recommend a Q-learning approach with two separate Q-functions (one for the hedging cost and one for the expected square of the hedging cost), but this example uses instead a simplified reward function.

Define Training Parameters

Next, specify an at-the-money option with three months to maturity is hedged. For simplicity, both the interest rate and dividend yield are set to `0`.

```
% Option parameters
Strike = 100;
Maturity = 21*3/250;

% Asset parameters
SpotPrice = 100;
ExpVol = .2;
ExpReturn = .05;

% Simulation parameters
rfRate = 0;
dT = 1/250;
nSteps = Maturity/dT;
nTrials = 5000;

% Transacation cost and cost function parameters
c = 1.5;
kappa = .01;
InitPosition = 0;

% Set the random generator seed for reproducibility.
rng(3)
```

Define Environment

In this section, the action and observation parameters, `actInfo` and `obsInfo`. The agent action is the current `hedge` value which can range between `0` and `1`. There are three variables in the agent observation:

- Moneyness (ratio of the spot price to the strike price)
- Time to maturity
- Position or amount of the underlying asset that is held

```
ObservationInfo             = rlNumericSpec([3 1],'LowerLimit',0,'UpperLimit',[10 Maturity 1]');
ObservationInfo.Name        = 'Hedging State';
ObservationInfo.Description = ['Moneyness', 'TimeToMaturity','Position'];

ActionInfo = rlNumericSpec([1 1],'LowerLimit',0,'UpperLimit',1);
ActionInfo.Name = 'Hedge';
```

Define Reward

From Cao [2 on page 10-47], the accounting P&L formulation and rewards (negative costs) are

$$R_{i+1} = V_{i+1} - V_i + H_{i+1}(S_{i+1} - S_i) - \kappa|S_{i+1}(H_{i+1} - H_i)|$$

where

R_i: Reward

V_i: Value of option

S_i: Spot price of underlying asset

H_i: Holding

κ: Transaction costs

A final reward at the last time step liquidates the hedge that is $\kappa|S_n(H_n)|$.

In this implementation, the reward (R_i) is penalized by the square of the reward multiplied by a constant to punish large swings in the value of the hedged position:

$$R_{i+1} = R_{i+1} - c(R_{i+1})^2$$

The reward is defined in `stepFcn` which is called at each step of the simulation.

```
env = rlFunctionEnv(ObservationInfo,ActionInfo, ...
    @(Hedge, LoggedSignals) stepFcn(Hedge,LoggedSignals,rfRate,ExpVol,dT,Strike,ExpReturn,c,kappa
    @() resetFcn(SpotPrice/Strike,Maturity,InitPosition));

obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

Create Environment Interface for RL Agent

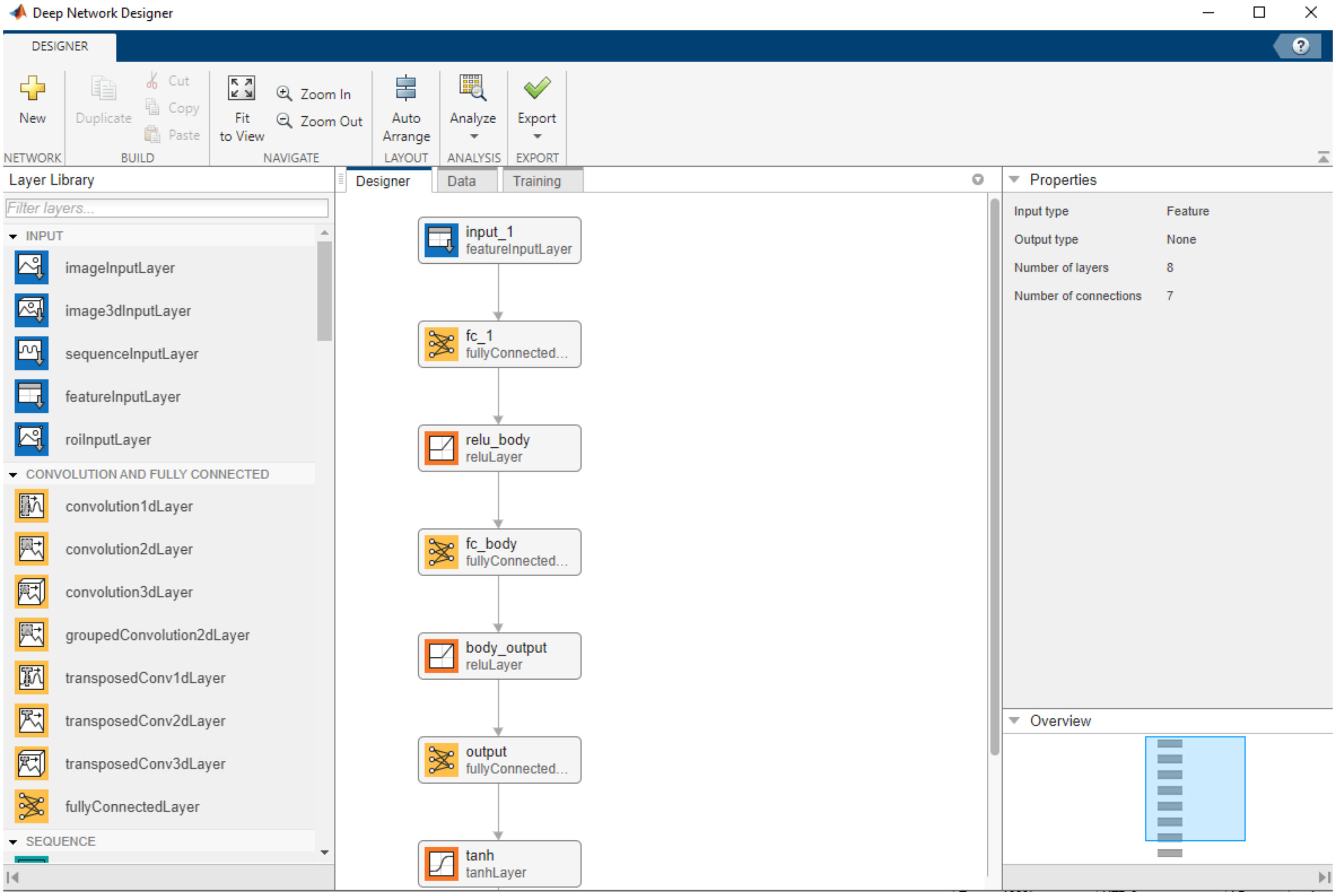
Create the DDPG agent using `rlDDPGAgent` (Reinforcement Learning Toolbox). While it is possible to create custom actor and critic networks, this example uses the default networks.

```
initOpts = rlAgentInitializationOptions('NumHiddenUnit',64);
criticOpts = rlOptimizerOptions("LearnRate",1e-4);
actorOpts = rlOptimizerOptions("LearnRate",1e-4);

agentOptions = rlDDPGAgentOptions(...
    "ActorOptimizerOptions",actorOpts,...
    "CriticOptimizerOptions",criticOpts,...
    "DiscountFactor",.9995,...
    "TargetSmoothFactor",5e-4);
agent = rlDDPGAgent(obsInfo,actInfo,initOpts,agentOptions);
```

Visualize Actor and Critic Networks

Visualize the actor and critic networks using the Deep Network Designer.

```
deepNetworkDesigner(layerGraph(getModel(getActor(agent))))
```



Input type Feature
Output type None
Number of layers 8
Number of connections 7

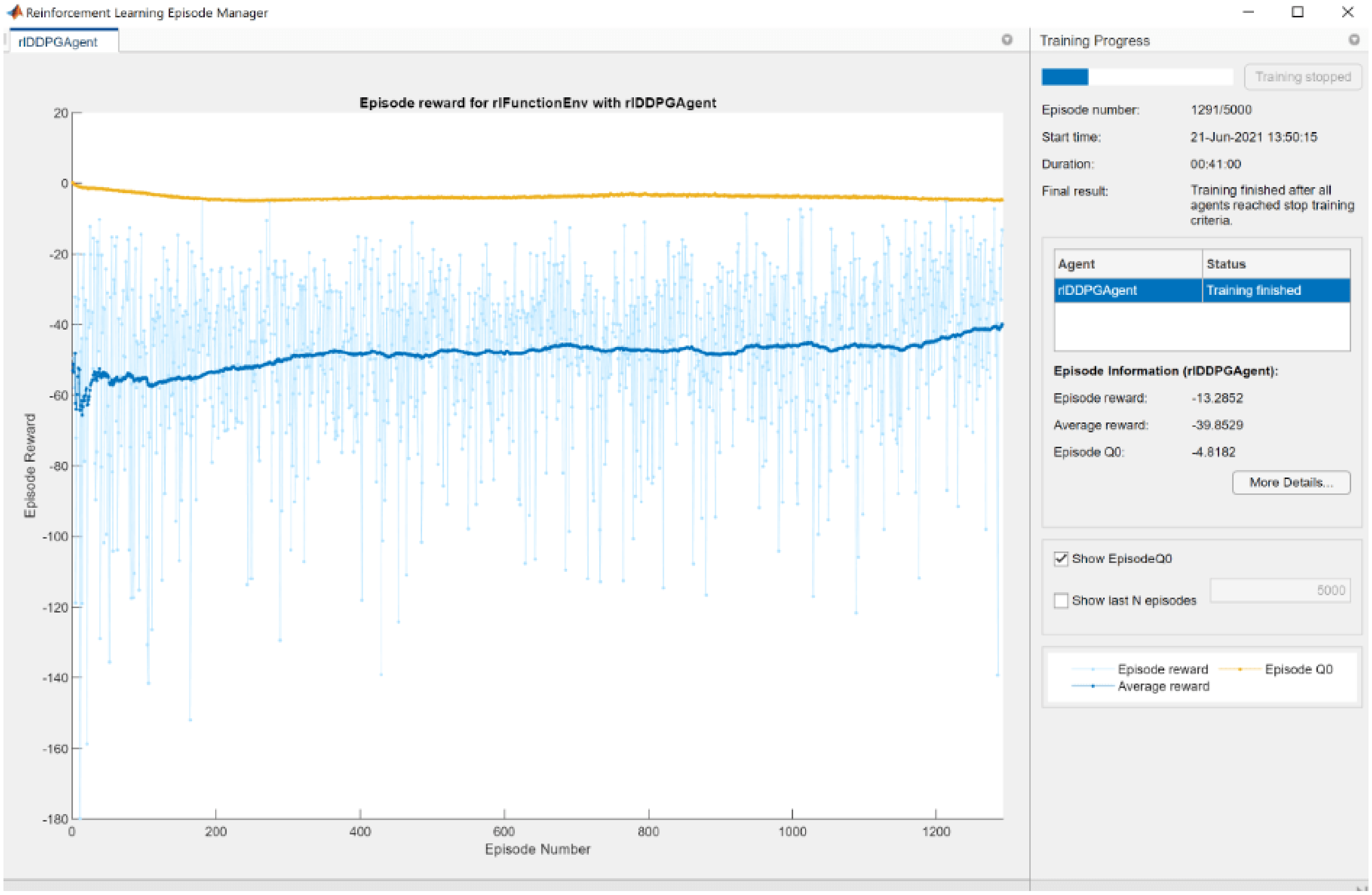
Train Agent

Train the agent using the `train` (Reinforcement Learning Toolbox) function.

```
trainOpts = rlTrainingOptions( ...
    'MaxEpisodes', nTrials, ...
    'MaxStepsPerEpisode',nSteps, ...
    'Verbose', false, ...
    'ScoreAveragingWindowLength',200,...
    'StopTrainingCriteria',"AverageReward", ...
    'StopTrainingValue',-40, ...
    'StopOnError',"on", ...
    "UseParallel",false);

doTraining = false;
if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainOpts);
else
```

```
    % Load the pretrained agent for the example.
    load('DeepHedgingDDPG.mat','agent')
end
```

To avoid waiting for the training, load pretrained networks by setting the `doTraining` flag to `false`. If you set `doTraining` to `true`, the Reinforcement Learning Episode Manager displays the training progress.



Episode number: 1291/5000
Start time: 21-Jun-2021 13:50:15
Duration: 00:41:00
Final result: Training finished after all agents reached stop training criteria.

Agent	Status
rlDDPGAgent	Training finished

Episode Information (rlDDPGAgent):
Episode reward: -13.2852
Average reward: -39.8529
Episode Q0: -4.8182

☑ Show EpisodeQ0
□ Show last N episodes 5000

Validate Agent

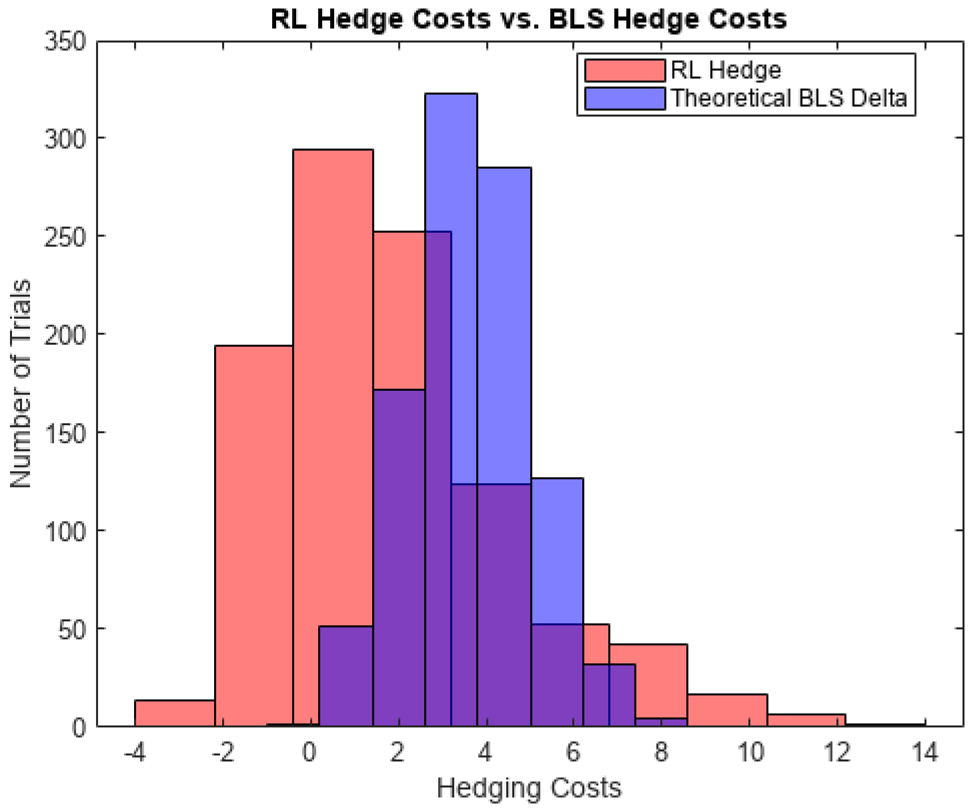
Use the Financial Toolbox™ functions `blsdelta` and `blsprice` for a conventional approach to calculate the price as a European call option. When comparing the conventional approach to the RL approach, the results are similar to the findings of Cao [2 on page 10-47] in Exhibit 4. This example demonstrates that the RL approach significantly reduces hedging costs.

```
% Simulation parameters
nTrials = 1000;

policy_BSM = @(mR,TTM,Pos) blsdelta(mR,1,rfRate,max(TTM,eps),ExpVol);

policy_RL = @(mR,TTM,Pos) arrayfun(@(mR,TTM,Pos) cell2mat(getAction(agent,[mR TTM Pos]')),mR,TTM

OptionPrice = blsprice(SpotPrice,Strike,rfRate,Maturity,ExpVol);

Costs_BSM = computeCosts(policy_BSM,nTrials,nSteps,SpotPrice,Strike,Maturity,rfRate,ExpVol,InitP
Costs_RL = computeCosts(policy_RL,nTrials,nSteps,SpotPrice,Strike,Maturity,rfRate,ExpVol,InitPos
HedgeComp = table(100*[-mean(Costs_BSM) std(Costs_BSM)]'/OptionPrice, ...
    100*[-mean(Costs_RL) std(Costs_RL)]'/OptionPrice, ...
    'RowNames',["Average Hedge Cost (% of Option Price)","STD Hedge Cost (% of Option Price)"],
    'VariableNames',["BSM","RL"]);
disp(HedgeComp)
```

```
                                               BSM       RL  
                                              ______    ______

    Average Hedge Cost (% of Option Price)    91.259    47.022
    STD Hedge Cost (% of Option Price)        35.712    68.119
```

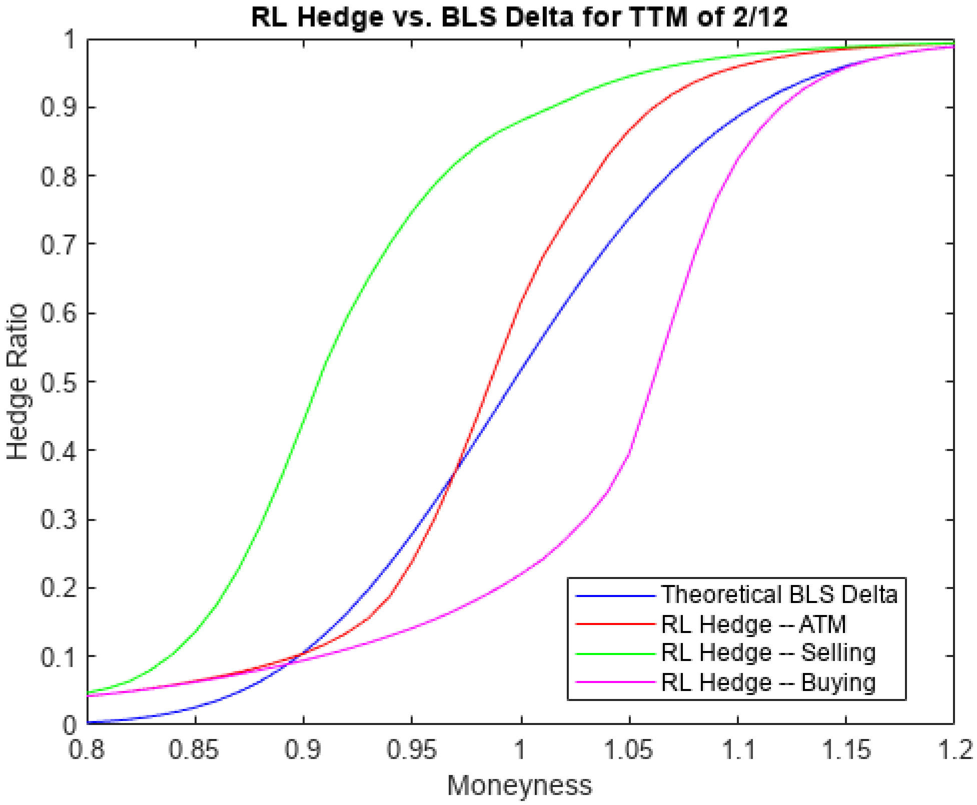
The following histogram shows the range of different hedging costs for both approaches. The RL approach performs better, but with a larger variance than the BSM approach. The RL appraoch in this example would likely benefit from the two Q-function approach that Cao [2 on page 10-47] discusses and implements.

```
figure
numBins = 10;
histogram(-Costs_RL,numBins,'FaceColor','r','FaceAlpha',.5)
hold on
histogram(-Costs_BSM,numBins,'FaceColor','b','FaceAlpha',.5)
xlabel('Hedging Costs')
ylabel('Number of Trials')
title('RL Hedge Costs vs. BLS Hedge Costs')
legend('RL Hedge','Theoretical BLS Delta','location','best')
```



A plot of the hedge Ratio with respect to moneyness shows the differences between the BSM and RL approaches. As discussed in Cao [2 on page 10-47], in the presence of transaction costs, the agent learns that "when delta hedging would require shares to be purchased, it tends to be optimal for a trader to be underhedged relative to delta. Similarly, when delta hedging would require shares to be sold, it tends to be optimal for a trader to be over-hedged relative to delta."

```
policy_RL_mR = @(mR,TTM,Pos) cell2mat(getAction(agent,[mR TTM Pos]'));

mRange = (.8:.01:1.2)';

figure
t_plot = 2/12;
plot(mRange,blsdelta(mRange,1,rfRate,t_plot,ExpVol),'b')
hold on
plot(mRange,arrayfun(@(mR) policy_RL_mR(mR,t_plot,blsdelta(mR,1,rfRate,t_plot,ExpVol)),mRange),'
plot(mRange,arrayfun(@(mR) policy_RL_mR(mR,t_plot,blsdelta(mR+.1,1,rfRate,t_plot,ExpVol)),mRange
plot(mRange,arrayfun(@(mR) policy_RL_mR(mR,t_plot,blsdelta(mR-.1,1,rfRate,t_plot,ExpVol)),mRange
legend('Theoretical BLS Delta','RL Hedge -- ATM','RL Hedge -- Selling','RL Hedge -- Buying', ...
    'location','best')
xlabel('Moneyness')
ylabel('Hedge Ratio')
title('RL Hedge vs. BLS Delta for TTM of 2/12')
```



References

[1] Buehler H., L. Gonon, J. Teichmann, and B. Wood. "Deep hedging." *Quantitative Finance*. Vol. 19, No. 8, 2019, pp. 1271–91.

[2] Cao J., J. Chen, J. Hull, and Z. Poulos. "Deep Hedging of Derivatives Using Reinforcement Learning." *The Journal of Financial Data Science*. Vol. 3, No. 1, 2021, pp. 10–27.

[3] Halperin I. "QLBS: Q-learner in the Black-Scholes (-Merton) Worlds." *The Journal of Derivatives*. Vol. 28, No. 1, 2020, pp. 99–122.

[4] Kolm P.N. and G. Ritter. "Dynamic Replication and Hedging: A Reinforcement Learning Approach." *The Journal of Financial Data Science*. Vol. 1, No. 1, 2019, pp. 159–71.

Local Functions

```
function [InitialObservation,LoggedSignals] = resetFcn(Moneyness,TimeToMaturity,InitPosition)
% Reset function to reset at the beginning of each episode.

LoggedSignals.State = [Moneyness TimeToMaturity InitPosition]';
InitialObservation  = LoggedSignals.State;

end

function [NextObs,Reward,IsDone,LoggedSignals] = stepFcn(Position_next,LoggedSignals,r,vol,dT,X,r
% Step function to evaluate at each step of the episode.

Moneyness_prev = LoggedSignals.State(1);
TTM_prev = LoggedSignals.State(2);
Position_prev = LoggedSignals.State(3);

S_prev = Moneyness_prev*X;

% GBM Motion
S_next = S_prev*((1 + mu*dT) + (randn* vol).*sqrt(dT));
TTM_next = max(0,TTM_prev - dT);

IsDone = TTM_next < eps;

stepReward = (S_next - S_prev)*Position_prev - abs(Position_next - Position_prev)*S_next*kappa -
    blsprice(S_next,X,r,TTM_next,vol) + blsprice(S_prev,X,r,TTM_prev,vol);

if IsDone
    stepReward = stepReward - Position_next*S_next*kappa;
end

Reward = stepReward - c*stepReward.^2;

LoggedSignals.State = [S_next/X;TTM_next;Position_next];
NextObs = LoggedSignals.State;

end

function perCosts = computeCosts(policy,nTrials,nSteps,SpotPrice,Strike,T,r,ExpVol,InitPos,dT,mu
% Helper function to compute costs for any hedging approach.

rng(0)

simOBJ = gbm(mu,ExpVol,'StartState',SpotPrice);
[simPaths,simTimes] = simulate(simOBJ,nSteps,'nTrials',nTrials,'deltaTime',dT);
simPaths = squeeze(simPaths);
```

```
rew = zeros(nSteps,nTrials);

Position_prev = InitPos;
Position_next = policy(simPaths(1,:)/Strike,T*ones(1,nTrials),InitPos*ones(1,nTrials));
for timeidx=2:nSteps+1
    rew(timeidx-1,:) = (simPaths(timeidx,:) - simPaths(timeidx-1,:)).*Position_prev - ...
        abs(Position_next - Position_prev).*simPaths(timeidx,:)*kappa - ...
        blsprice(simPaths(timeidx,:),Strike,r,max(0,T - simTimes(timeidx)),ExpVol) + ...
        blsprice(simPaths(timeidx-1,:),Strike,r,T - simTimes(timeidx-1),ExpVol);

    if timeidx == nSteps+1
         rew(timeidx-1,:) =  rew(timeidx-1,:) - Position_next.*simPaths(timeidx,:,:)*kappa;
    else
        Position_prev = Position_next;
        Position_next = policy(simPaths(timeidx,:,:)/Strike,(T - simTimes(timeidx)).*ones(1,nTria
    end
end

perCosts = sum(rew);

end
```

See Also

blsprice | blsdelta | rlDDPGAgent

External Websites

- Reinforcement Learning in Finance (4 min 15 sec)

Using Financial Timetables

Convert Financial Time Series Objects (fints) to Timetables

In this section...

In R2023a, financial time series (`fints`), and its associated methods have been removed and are replaced with a MATLAB `timetable` function. If you use `fints` or the associated methods, you receive an error. To help you convert from the older `fints` to the newer `timetable` functionality, use the following information.

Create Time Series

I/O Related Operations

Task	Removed Functionality	Use This Functionality
Construct by passing in data and dates	`fints(dates,data,datanames)`	Use `timetable`: `timetable(rowTimes,var1,...,varN,'VariableNames',{'a','b',...})`

Construct by conversion of files	`ascii2fts(filename,descrow,colheadrow,skiprows)`	Use `readtable` and `table2timetable`:
		`T = readtable(filename,opts,Name,Value)`
		`TT = table2timetable(T,'RowTimes',timeVarName)`

Write files	`fts2ascii(filename,tsobj,exttext)`	Use `writetable`: `writetable(TT,filename)`

Convert to matrix	`fts2mat(tsobj)`	`S = vartype('numeric');`
		`TT2 = TT(:,S)`
		`TT2.Variables`

Index an Object

Indexing an Object

Task	Removed Functionality	Use This Functionality
Indexing with a date	`myfts('05/11/99')`	`TT({'1999-05-11'},:)`

Indexing with a date range	`myfts ('05/11/99::05/15/99')`	Use `timerange`:
		`S = timerange('1999-05-11','1999-05-15');`
		`TT2 = TT(S,:)`

Indexing with integers for rows	`myfts.series2(1)`
	`myfts.series2([1, 3, 5])`
	`myfts.series2(16:20)`
		`TT(1,{'series2'})`
		`TT([1, 3, 5],{'series2'})`
		`TT(16:20,{'series2'})`

Contents of a specific time field	`myfts.times`	Use `timeofday`: `timeofday(TT.Properties.RowTimes)`
Contents for a specific field in a matrix	`fts2mat(myfts.series2)`	`TT.series2`

Transform Time Series

Assume that all variables are numeric within a timetable, or the operations can be applied on TT2:

```
S = vartype('numeric');
```

```
TT2 = TT(:,S)
```

Filter Time Series

Task	Removed Functionality	Use This Functionality
Boxcox transformation	`newfts = boxcox(oldfts)`	Use `boxcox`: `TT.Variables = boxcox(TT.Variables)`

Differencing	`diff(myfts)`	`TT{2:end,:} = diff(TT.Variables)`
		`TT(1,:) = []`

Indexing with integers for rows	`fillts(oldfts,fill_method)`	Use `fillmissing`: `fillmissing(TT,method)` (Assumes no missing dates)
Linear filtering	`filter(B,A, myfts)`	Use `filter`: `TT.Variables = filter(b,a,TT.Variables)`

Lag or lead time series object	`lagts(myfts,lagperiod)`
	`leadts(myfts,leadperiod)`
		Use `lag`:
		`lag(TT,lagperiod)`
		`lag(TT,-leadperiod)`
		(Assumes a regularly spaced timetable)

Periodic average	`peravg(myfts)`	Use `retime`: `retime(TT,newTimes,'mean')`
Downsample data	`resamplets(oldfts,samplestep)`	Use `retime`: `retime(TT,newTimeStep,method)`
Smooth data	`smoothts(input)`	Use `smoothdata`: `smoothdata(TT)`
Moving average	`tsmovavg(tsobj,method,lag)`	Use `movavg`: `movavg(TT,type,windowSize)`

Convert Time Series

Assume that all variables are numeric within a timetable, or the operations can be applied on TT2:

```
S = vartype('numeric');
```

```
TT2 = TT(:,S)
```

Conversion Operations

Task	Removed Functionality	Use This Functionality
Convert to specified frequency	`convertto(oldfts,newfreq)`	Use `retime`: `retime(TT,newTimeStep,method)`
Convert to annual	`toannual(oldfts,...)`	Use `convert2annual`: `convert2annual(TT,…)`
Convert to daily	`todaily(oldfts,...)`	Use `convert2daily`: `convert2daily(TT,…)`
Convert to monthly	`tomonthly(oldfts,...)`	Use `convert2monthly`: `convert2monthly(TT,…)`
Convert to quarterly	`toquarterly(oldfts,...)`	Use `convert2quarterly`: `convert2quarterly(TT,…)`
Convert to semiannual	`tosemi(oldfts,...)`	Use `convert2semiannual`: `convert2semiannual(TT,…)`
Convert to weekly	`toweekly(oldfts,...)`	Use `convert2weekly`: `convert2weekly(TT,…)`

Merge Time Series

Merge Operations

Task	Removed Functionality	Use This Functionality
Merge multiple time series objects	`merge(fts1,fts2)`	`[TT1;TT2]` (requires variable name to be the same) `unique(TT)`
Concatenate financial time series objects horizontally	`horzcat(fts1,fts2)` or `[fts1,fts2]`	Use `horzcat` or `synchronize`: `horzcat[TT1,TT2]` (requires variable name to be the same) or `synchronize(TT1,TT2)`
Concatenate financial time series objects vertically	`vertcat(fts1,fts2)` or `[fts1;fts2]`	Use `vertcat`: `vertcat[TT1;TT2]`

Analyze Time Series

Due to flexibility of a timetable that can hold heterogeneous variables, a timetable does not support math operations or descriptive statistical calculations. If you would like to apply any numeric calculations on a timetable, use the following guidelines.

Assume that all variables are numeric within a timetable, or the operations can be applied on `TT2`:

```
S = vartype('numeric');
```

```
TT2 = TT(:,S)
```

Descriptive Statistics and Arithmetic and Math Operations

Task	Removed Functionality	Use This Functionality
Extract out numerical data	`srs2 = myfts.series2`	`TT.Variables`

Apply some options (statistics)	For example: `min`, `max`, `mean`, `median`, `cov`, `std`, and `var`	Use `cov` (or `max`, `mean`, `median`, `min`, `std`, or `var`):
		`cov(TT.Variables)`

Apply some options (operations)	For example: `sum` and `cumsum`	Use `sum` or `cumsum`: `TT.Variables = cumsum(TT.Variables)`

Data Extraction

Refer to `timetable` documentation for data extraction methods and examples.

See Also

`timetable` | `retime` | `synchronize` | `timerange` | `withtol` | `vartype` | `issorted` | `sortrows` | `unique` | `diff` | `isregular` | `rmmissing` | `fillmissing` | `convert2daily` | `convert2weekly` | `convert2monthly` | `convert2quarterly` | `convert2semiannual` | `convert2annual`

Related Examples

- “Use Timetables in Finance” on page 11-7
- “Select Times in Timetable”
- “Resample and Aggregate Data in Timetable”
- “Combine Timetables and Synchronize Their Data”
- “Retime and Synchronize Timetable Variables Using Different Methods”
- “Clean Timetable with Missing, Duplicate, or Nonuniform Times”

Use Timetables in Finance

Use timetables to visualize and calculate weekly statistics from simulated daily stock data.

Step 1. Load the data.

The data for this example is in the MAT-file SimulatedStock.mat, which loads the following:

- Dates corresponding to the closing stock prices, TMW_DATES
- Opening stock prices, TMW_OPEN
- Daily high of stock prices, TMW_HIGH
- Daily low of stock prices, TMW_LOW
- Closing stock prices, TMW_CLOSE, TMW_CLOSE_MISSING
- Daily volume of traded, TMW_VOLUME
- Data in a table, TMW_TB

```
load SimulatedStock.mat TMW_*
```

Step 2. Create timetables.

In timetables, you can work with financial time series rather than with vectors. When using a timetable, you can easily track the dates. You can manipulate the data series based on the dates, because a timetable object tracks the administration of a time series.

Use the MATLAB® timetable function to create a timetable object. Alternatively, you can use the MATLAB conversion function table2timetable to convert a table to a timetable. In this example, the timetable TMW_TT is constructed from a table and is only for illustration purposes. After you create a timetable object, you can use the Description field of the timetable object to store meta-information about the timetable.

```
% Create a timetable from vector input
TMW = timetable(TMW_OPEN,TMW_HIGH,TMW_LOW,TMW_CLOSE_MISSING,TMW_VOLUME, ...
    'VariableNames',{'Open','High','Low','Close','Volume'},'RowTimes',TMW_DATES);

% Convert from a table to a timetable
TMW_TT = table2timetable(TMW_TB,'RowTimes',TMW_DATES);

TMW.Properties.Description = 'Simulated stock data.';

TMW.Properties
```

```
ans = 
  TimetableProperties with properties:

             Description: 'Simulated stock data.'
                UserData: []
          DimensionNames: {'Time'  'Variables'}
           VariableNames: {'Open'  'High'  'Low'  'Close'  'Volume'}
    VariableDescriptions: {}
           VariableUnits: {}
      VariableContinuity: []
                RowTimes: [1000x1 datetime]
               StartTime: 04-Sep-2012
```

```
              SampleRate: NaN
                TimeStep: NaN
                  Events: []
        CustomProperties: No custom properties are set.
      Use addprop and rmprop to modify CustomProperties.
```

Step 3. Calculate basic data statistics, and fill the missing data.

Use the MATLAB `summary` function to view basic statistics of the `timetable` data. By reviewing the summary for each variable, you can identify missing values. You can then use the MATLAB `fillmissing` function to fill in missing data in a timetable by specifying a fill method.

```
summaryTMW = summary(TMW);
summaryTMW.Close
```

```
ans = struct with fields:
           Size: [1000 1]
           Type: 'double'
    Description: ''
          Units: ''
     Continuity: []
            Min: 83.4200
         Median: 116.7500
            Max: 162.1100
     NumMissing: 3
```

```
TMW = fillmissing(TMW,'linear');
summaryTMW = summary(TMW);
summaryTMW.Close
```

```
ans = struct with fields:
           Size: [1000 1]
           Type: 'double'
    Description: ''
          Units: ''
     Continuity: []
            Min: 83.4200
         Median: 116.7050
            Max: 162.1100
     NumMissing: 0
```

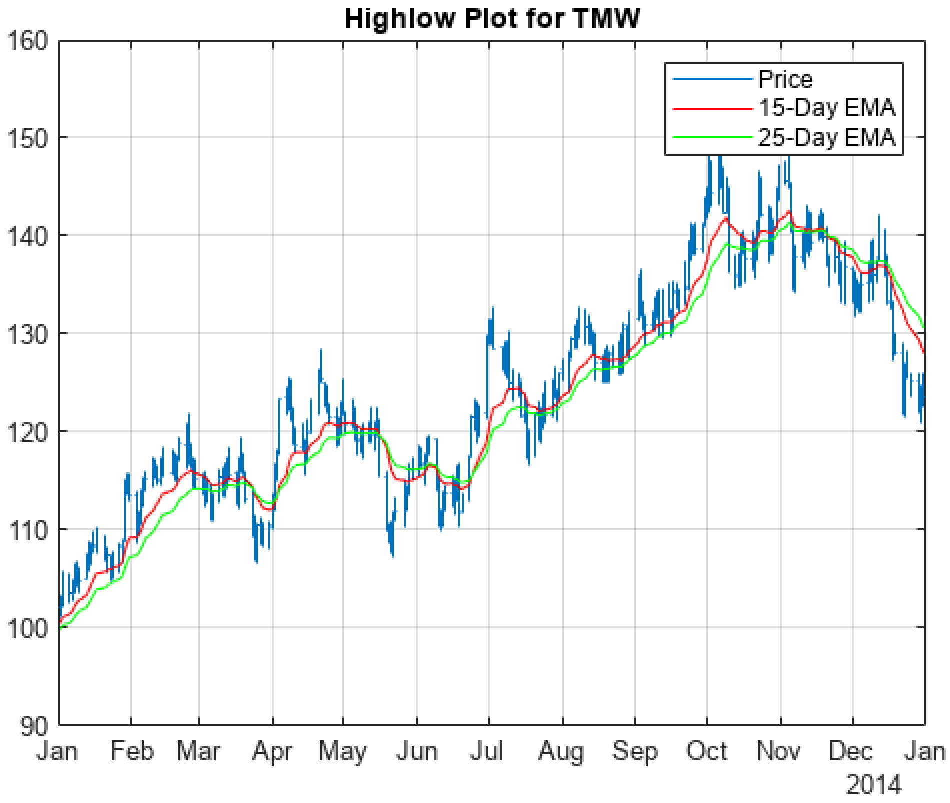
```
summaryTMW.Time
```

```
ans = struct with fields:
          Size: [1000 1]
          Type: 'datetime'
           Min: 04-Sep-2012
        Median: 31-Aug-2014
           Max: 24-Aug-2016
    NumMissing: 0
      TimeStep: NaN
```

Step 4. Visualize the data.

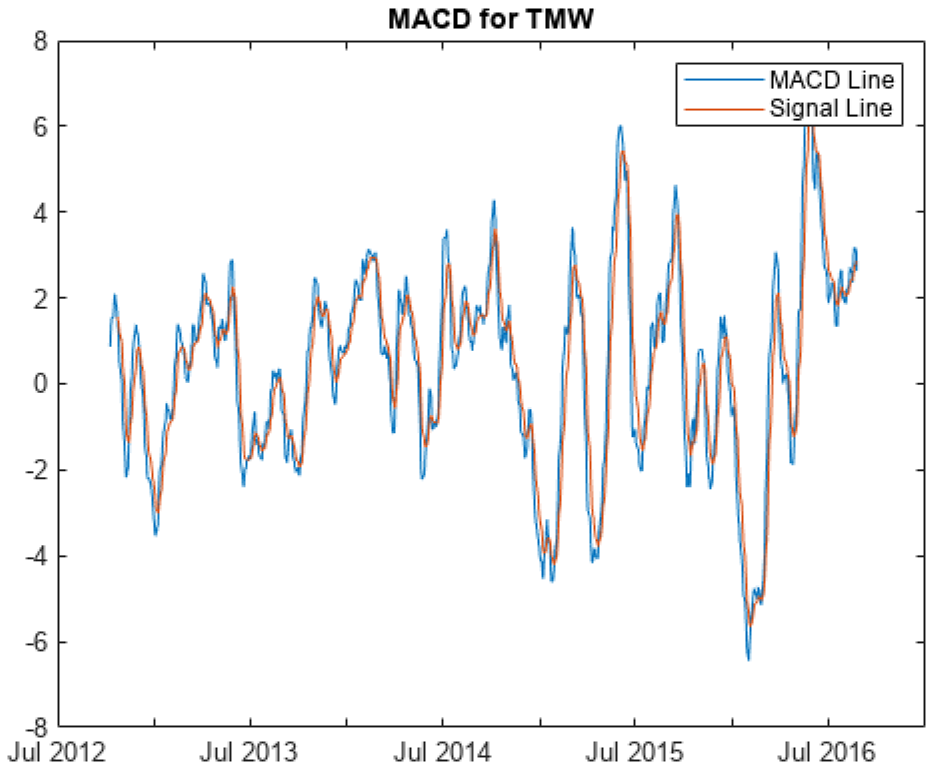
To visualize the timetable data, use financial charting functions such as `highlow` or `movavg`. For this example, the moving average information is plotted on the same chart for `highlow` to provide a complete visualization. To obtain the stock performance in 2014, use the MATLAB `timerange` function to select rows of the `timetable`. To visualize a technical indicator such as the Moving Average Convergence Divergence (MACD), pass the `timetable` object into the `macd` function for analysis.

```
index = timerange(datetime('01-Jan-2014','Locale','en_US'),datetime('31-Dec-2014','Locale','en_US

highlow(TMW(index,:));
hold on

ema15 = movavg(TMW(:,'Close'),'exponential',15);
ema25 = movavg(TMW(:,'Close'),'exponential',25);

ema15 = ema15(index,:);
ema25 = ema25(index,:);
plot(ema15.Time,ema15.Close,'r');
plot(ema25.Time,ema25.Close,'g');
hold off

legend('Price','15-Day EMA','25-Day EMA')
title('Highlow Plot for TMW')
```



```
[macdLine, signalLine] = macd(TMW(:,'Close'));

plot(macdLine.Time,macdLine.Close);
hold on
plot(signalLine.Time,signalLine.Close);
```

```
hold off

title('MACD for TMW')
legend('MACD Line', 'Signal Line')
```



Step 5. Create a weekly return and volatility series.

To calculate weekly return from the daily stock prices, you must resample the data frequency from daily to weekly. When working with timetables, use the MATLAB functions `retime` or `synchronize` with various aggregation methods to calculate weekly statistics. To adjust the timetable data to a time-vector basis, use `retime` and use `synchronize` with multiple timetables.

```
weeklyOpen = retime(TMW(:,'Open'),'weekly','firstvalue');
weeklyHigh = retime(TMW(:,'High'),'weekly','max');
weeklyLow = retime(TMW(:,'Low'),'weekly','min');
weeklyClose = retime(TMW(:,'Close'),'weekly','lastvalue');
weeklyTMW = [weeklyOpen,weeklyHigh,weeklyLow,weeklyClose];

weeklyTMW = synchronize(weeklyTMW,TMW(:,'Volume'),'weekly','sum');
head(weeklyTMW)
```

```
       Time         Open      High      Low      Close      Volume  
    ___________    ______    ______    _____    _____    __________

    02-Sep-2012       100    102.38    98.45    99.51    2.7279e+07
    09-Sep-2012     99.72    101.55    96.52    97.52    2.8518e+07
    16-Sep-2012     97.35     97.52     92.6    93.73    2.9151e+07
    23-Sep-2012     93.55     98.03    92.25    97.35     3.179e+07
    30-Sep-2012      97.3     103.15      96.68      99.66     3.3761e+07
    07-Oct-2012     99.76     106.61       98.7     104.23     3.1299e+07
    14-Oct-2012    104.54     109.75     100.55     103.77     3.1534e+07
    21-Oct-2012    103.84     104.32      96.95      97.41     3.1706e+07
```

To perform calculations on entries in a `timetable`, use the MATLAB `rowfun` function to apply a function to each row of a weekly frequency timetable.

```
returnFunc = @(open,high,low,close,volume) log(close) - log(open);
weeklyReturn = rowfun(returnFunc,weeklyTMW,'OutputVariableNames',{'Return'});

weeklyStd = retime(TMW(:,'Close'),'weekly',@std);
weeklyStd.Properties.VariableNames{'Close'} = 'Volatility';

weeklyTMW = [weeklyReturn,weeklyStd,weeklyTMW]
```

```
weeklyTMW=208×7 timetable
       Time           Return        Volatility     Open      High      Low      Close      Volume
    ___________    ___________    __________    ______    ______    ______    ______    _________

    02-Sep-2012      -0.004912      0.59386        100    102.38     98.45     99.51    2.7279e+0
    09-Sep-2012      -0.022309      0.63563      99.72    101.55     96.52     97.52    2.8518e+0
    16-Sep-2012      -0.037894      0.93927      97.35     97.52      92.6     93.73    2.9151e+0
    23-Sep-2012       0.039817       2.0156      93.55     98.03     92.25     97.35     3.179e+0
    30-Sep-2012       0.023965       1.1014       97.3    103.15     96.68     99.66    3.3761e+0
    07-Oct-2012       0.043833       1.3114      99.76    106.61      98.7    104.23    3.1299e+0
    14-Oct-2012     -0.0073929       1.8097     104.54    109.75    100.55    103.77    3.1534e+0
    21-Oct-2012      -0.063922       2.1603     103.84    104.32     96.95     97.41    3.1706e+0
    28-Oct-2012      -0.028309       0.9815      97.45      99.1     92.58     94.73    1.9866e+0
    04-Nov-2012    -0.00010566        1.224      94.65      96.1     90.82     94.64    3.5043e+0
    11-Nov-2012       0.077244       2.4854      94.39    103.98     93.84    101.97    3.0624e+0
    18-Nov-2012       0.022823      0.55896     102.23    105.27    101.24    104.59    2.5803e+0
    25-Nov-2012      -0.012789        1.337     104.66    106.02    100.85    103.33    3.1402e+0
    02-Dec-2012      -0.043801       0.2783     103.37    103.37     97.69     98.94    3.2136e+0
    09-Dec-2012      -0.063475       1.9826      99.02     99.09     91.34     92.93    3.4447e+0
    16-Dec-2012      0.0025787       1.2789      92.95      94.2     88.58     93.19    3.3247e+0
      ⋮
```

See Also

`timetable` | `retime` | `synchronize` | `timerange` | `withtol` | `vartype` | `issorted` | `sortrows` | `unique` | `diff` | `isregular` | `rmmissing` | `fillmissing`

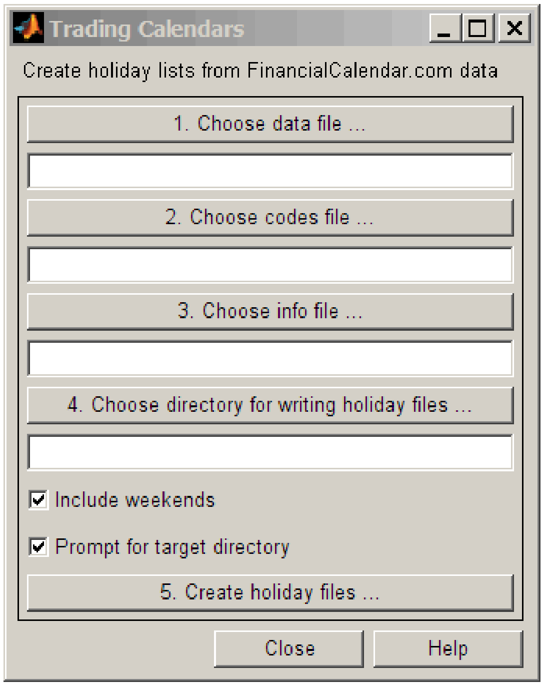
Related Examples

- “Select Times in Timetable”
- “Resample and Aggregate Data in Timetable”
- “Combine Timetables and Synchronize Their Data”
- “Retime and Synchronize Timetable Variables Using Different Methods”
- “Clean Timetable with Missing, Duplicate, or Nonuniform Times”
- “Convert Financial Time Series Objects (fints) to Timetables” on page 11-2

Trading Date Utilities

Trading Calendars User Interface

Use the `createholidays` function to open the Trading Calendars user interface.



Trading Calendars

Create holiday lists from FinancialCalendar.com data

☑ Include weekends

☑ Prompt for target directory

The `createholidays` function supports `https://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 `https://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 12-4

UICalendar User Interface

In this section...
"Using UICalendar in Standalone Mode" on page 12-4
"Using UICalendar with an Application" on page 12-4

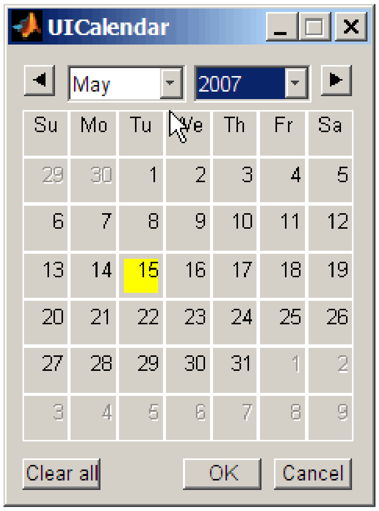
Using UICalendar in Standalone Mode

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:



May 2007

Su	Mo	Tu	We	Th	Fr	Sa
29	30	1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31	1	2
3	4	5	6	7	8	9

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 starts up 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:

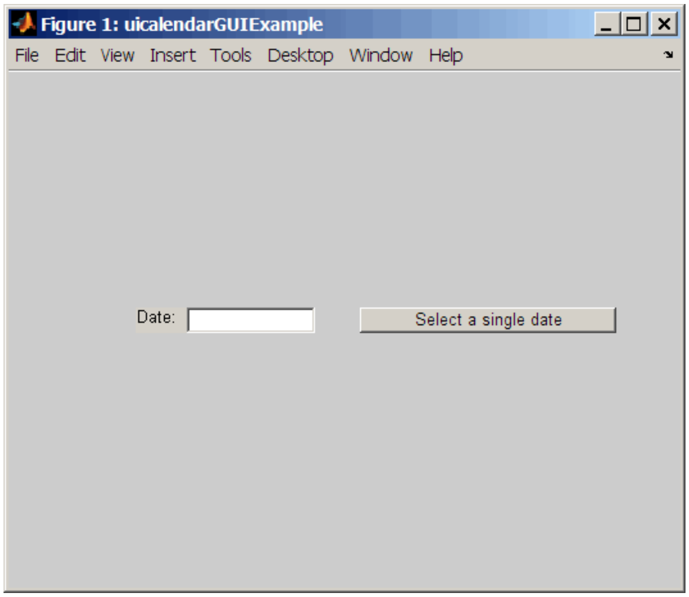


Figure 1: uicalendarGUIExample
File Edit View Insert Tools Desktop Window Help
Date:
Select a single date

7 Click **Select a single date** to display the UICalendar user interface:

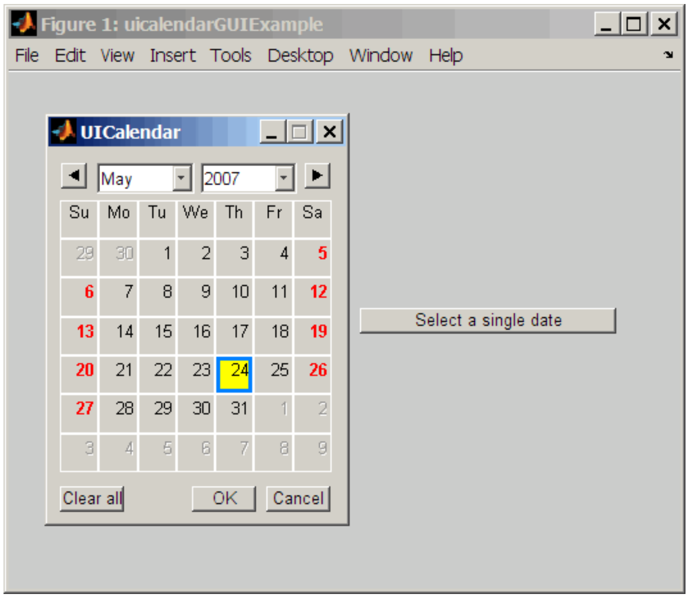


Figure 1: uicalendarGUIExample
File Edit View Insert Tools Desktop Window Help
UICalendar
May
2007
Clear all
OK
Cancel
Select a single date

Su	Mo	Tu	We	Th	Fr	Sa
29	30	1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31	1	2
3	4	5	6	7	8	9

8 Select a date and click **OK** to display the date in the text field:

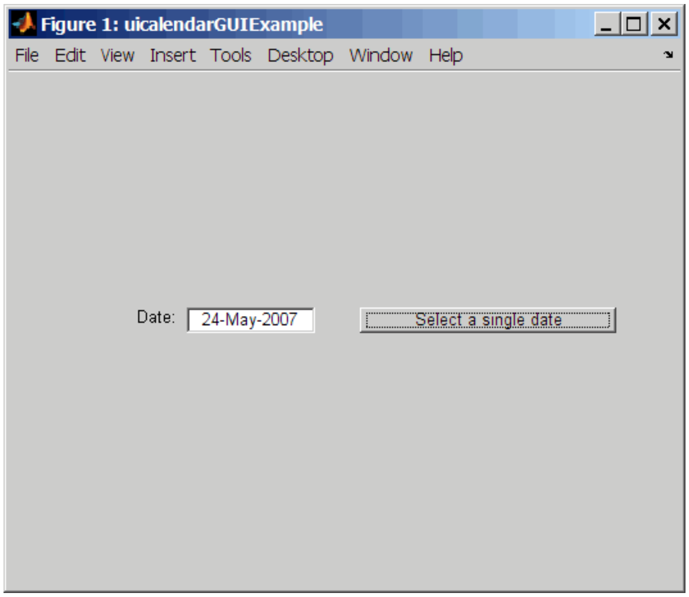


Figure 1: uicalendarGUIExample
File Edit View Insert Tools Desktop Window Help
Select a single date

Date: 24-May-2007

See Also

`createholidays` | `holidays` | `nyseclosures`

Related Examples

- “Trading Calendars User Interface” on page 12-2
- “Handle and Convert Dates” on page 2-2

13

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
`stochosc`	Fast stochastics, slow statistics
`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

adosc | chaikosc | macd | stochosc | tsaccel | tsmom | chaikvolat | willpctr | negvolidx | posvolidx | rsindex | adline | bollinger | hhigh | llow | medprice | onbalvol | prcroc | pvtrend | typprice | volroc | wclose | willad

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 standard Monte Carlo or Quasi-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 14-28	• Geometric Brownian Motion (GBM) on page 14-22 • Constant Elasticity of Variance (CEV) on page 14-22 • Stochastic Differential Equation (SDE) on page 14-14 • Stochastic Differential Equations from Drift and Diffusion Objects (SDEDDO) on page 14-16 • Stochastic Differential Equations from Linear Drift (SDELD) on page 14-19 • Heston Stochastic Volatility (Heston) on page 14-26
“Simulating Interest Rates” on page 14-48	• Hull-White-Vasicek (HWV) on page 14-25 • Cox-Ingersoll-Ross (CIR) on page 14-24 • Stochastic Differential Equation (SDE) on page 14-14 • Stochastic Differential Equations from Drift and Diffusion Objects (SDEDDO) on page 14-16 • Stochastic Differential Equations from Mean-Reverting Drift (SDEMRD) Models on page 14-23
“Pricing Equity Options” on page 14-45	Geometric Brownian Motion (GBM) on page 14-22
“Stratified Sampling” on page 14-57	All supported models on page 14-11
“Quasi-Monte Carlo Simulation” on page 14-62	All supported models on page 14-11

To perform this task ...	Use these types of models ...
“Performance Considerations” on page 14-64	All supported models on page 14-11

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

NTrials, NPeriods, and NSteps

SDE functions 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 14-65 and “Managing Memory” on page 14-64.

See Also

`sde` | `bm` | `gbm` | `bates` | `merton` | `drift` | `diffusion` | `sdeddo` | `sdeld` | `cev` | `cir` | `heston` | `hwv` | `sdemrd` | `ts2func` | `simulate` | `simByQuadExp` | `simByEuler` | `simBySolution` | | `interpolate`

Related Examples

- “Base SDE Models” on page 14-14
- “Drift and Diffusion Models” on page 14-16

- “Linear Drift Models” on page 14-19
- “Parametric Models” on page 14-21

More About

- “SDE Class Hierarchy” on page 14-5
- “SDE Models” on page 14-7

SDE Class Hierarchy

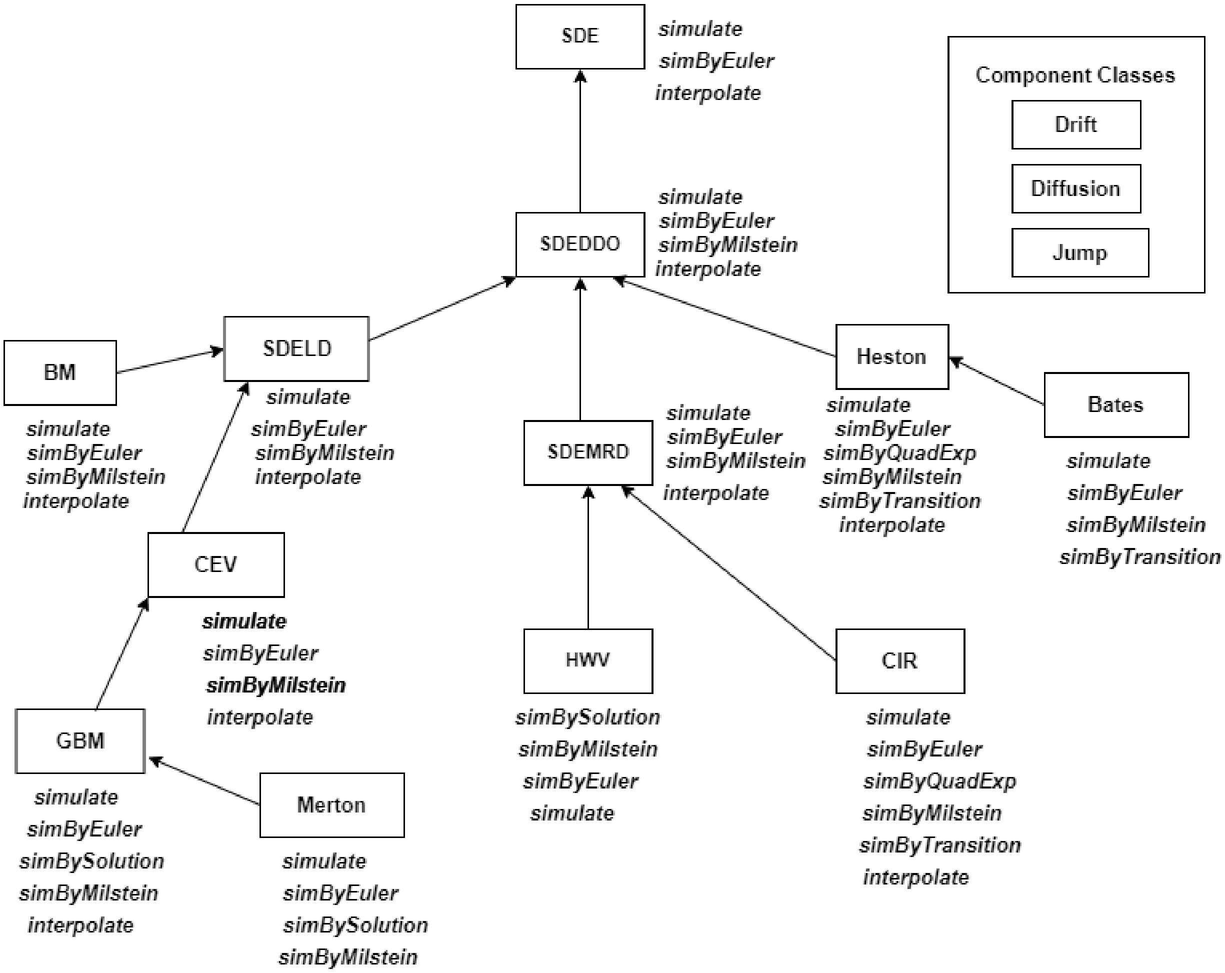
The Financial Toolbox SDE class structure represents a generalization and specialization hierarchy. The top-level class provides the most general model interface and offers the default Monte Carlo simulation and interpolation methods. In turn, derived classes offer restricted interfaces that simplify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

The following table lists the SDE classes. The introductory examples in “Available Models” on page 14-11 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 14-14
`Drift`, `Diffusion`	`drift`, `diffusion`, and “Overview” on page 14-16
`SDEDDO`	`sdeddo` and “Drift and Diffusion Models” on page 14-16
`SDELD`	`sdeld` and “Linear Drift Models” on page 14-19
`CEV`	`cev` and “Creating Constant Elasticity of Variance (CEV) Models” on page 14-22
`BM`	`bm` and “Creating Brownian Motion (BM) Models” on page 14-21
`SDEMRD`	`sdemrd` and “Creating Stochastic Differential Equations from Mean-Reverting Drift (SDEMRD) Models” on page 14-23
`GBM`	`gbm` and “Creating Geometric Brownian Motion (GBM) Models” on page 14-22
`HWV`	`hwv` and “Creating Hull-White/Vasicek (HWV) Gaussian Diffusion Models” on page 14-25
`CIR`	`cir` and “Creating Cox-Ingersoll-Ross (CIR) Square Root Diffusion Models” on page 14-24
`Heston`	`heston` and “Creating Heston Stochastic Volatility Models” on page 14-26
`Merton`	`merton`
`Bates`	`bates`

The following figure illustrates the inheritance relationships among SDE classes.



See Also

`sde` | `bm` | `gbm` | `bates` | `merton` | `drift` | `diffusion` | `sdeddo` | `sdeld` | `cev` | `cir` | `heston` | `hwv` | `sdemrd` | `ts2func` | `simulate` | `simByEuler` | `simByQuadExp` | `simBySolution` | `simBySolution` | `interpolate`

Related Examples

- “Base SDE Models” on page 14-14
- “Drift and Diffusion Models” on page 14-16
- “Linear Drift Models” on page 14-19
- “Parametric Models” on page 14-21

More About

- “SDEs” on page 14-2
- “SDE Models” on page 14-7

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`, `diffusion` objects are used in model specification, but neither of these types of objects in and of themselves makes up a complete model. Usually, you do not need to create `drift`, `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

Creating Objects

For examples and more information on creating SDE objects, see:

- “Available Models” on page 14-11
- “Simulating Equity Prices” on page 14-28
- “Simulating Interest Rates” on page 14-48

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 valid:

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

  ```
  obj.A = 3
  ```
- Objects do not support partial parameter assignment as data structures do. Therefore, statements like the following are invalid:

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

Creating and Evaluating Models

- You can create objects of any model class only if enough information is available to determine unambiguously the dimensionality of the model. Because each object offers 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 14-23.

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:

$$dX_t = F(t, X_t)dt + G(t, X_t)dW_t \tag{14-1}$$

where:

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

- G is an *NVars*-by-*NBrowns* matrix-valued diffusion-rate function.

The drift and diffusion rates, F and G, respectively, are general functions of a real-valued scalar sample time t and state vector X_t. Also, static (non-time-variable) coefficients are simply a special case of the more general dynamic (time-variable) situation, just as a function can be a trivial constant; for example, $f(t,X_t) = 4$. The SDE in “Equation 14-1” is useful in implementing derived classes that impose additional structure on the drift and diffusion-rate functions.

Specifying User-Defined Functions as Model Parameters

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

Model parameters that are specified as functions are evaluated in the same way as user-specified random number (noise) generation functions. (For more information, see “Evaluating Different Types of Functions” on page 14-9.) 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 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*, a simulation technique that provides a discrete-time approximation of a continuous-time stochastic process. 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 14-45.

Therefore, all simulation methods evaluate noise generation functions as:

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

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

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

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

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

$$F(t, X_t) = A(t) + B(t)X_t \qquad (14\text{-}2)$$

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

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

$$F(t, X_t) = S(t)[L(t) - X_t] \tag{14-3}$$

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

Similarly, consider the following diffusion-rate specification:

$$G(t, X_t) = D(t, X_t^{\alpha(t)})V(t) \tag{14-4}$$

where *D* is an *NVars*-by-*NVars* diagonal matrix-valued function. Each diagonal element of *D* is the corresponding element of the state vector raised to the corresponding element of an exponent *Alpha*, which is also an *NVars*-by-1 vector-valued function. *V* is an *NVars*-by-*NBrowns* matrix-valued function of instantaneous volatility rates. Each row of *V* corresponds to a particular state variable, and each column corresponds to a particular Brownian source of uncertainty. *V* associates the exposure of state variables with sources of risk.

The parametric specifications for the drift and diffusion-rate functions associate parametric restrictions with familiar models derived from the general SDE class, and provide coverage for many 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)	$dX_t = B(t)X_tdt + V(t)X_tdW_t$
Constant Elasticity of Variance (CEV)	$dX_t = B(t)X_tdt + V(t)X_t^{\alpha(t)}dW_t$
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	$dX_{1t} = B(t)X_{1t}dt + \sqrt{X_{2t}}X_{1t}dW_{1t}$ $dX_{2t} = S(t)[L(t) - X_{2t}]dt + V(t)\sqrt{X_{2t}}dW_{2t}$
Merton	$dX_t = B(t, X_t)X_tdt + D(t, X_t)V(t, x_t)dW_t + Y(t, X_t)X_tdN_t$
Bates	Bates models are bivariate composite models. Each Bates model consists of two coupled univariate models: • A geometric Brownian motion (`gbm`) model with a stochastic volatility function. $dX_{1t} = B(t)X_{1t}dt + \sqrt{X_{2t}}X_{1t}dW_{1t} + Y(t)X_{1t}dN_t$ • A Cox-Ingersoll-Ross (`cir`) square root diffusion model. $dX_{2t} = S(t)[L(t) - X_{2t}]dt + V(t)\sqrt{X_{2t}}dW_{2t}$

SDE Simulation and Interpolation 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)

You can use the `simBySolution` function to simulate approximate solutions of diagonal-drift processes for the following classes:

- `gbm` supports `simBySolution`
- `hwv` supports `simBySolution`
- `merton` supports `simBySolution`

You can use the `simBySolution` function with the name-value arguments for `MonteCarloMethod` and `QuasiSequence` to simulates approximate solutions of quasi-Monte Carlo simulations for the following classes:

- `gbm` supports `simBySolution`
- `hwv` supports `simBySolution`

- merton supports simBySolution

In addition, you can also use:

- simByTransition with a cir object to approximates a continuous-time Cox-Ingersoll-Ross (CIR) model by an approximation of the transition density function.
- simByTransition with a bates object to approximates a continuous-time Bates model by an approximation of the transition density function.
- simByTransition with a heston object to approximates a continuous-time Heston model by an approximation of the transition density function.
- simByQuadExp with a heston, bates, or cir object to generate sample paths by using a Quadratic-Exponential discretization scheme.

.

See Also

sde | bm | gbm | merton | bates | drift | diffusion | sdeddo | sdeld | cev | cir | heston | hwv | sdemrd | ts2func | simulate | simByEuler | simBySolution | simByQuadExp | simBySolution | interpolate

Related Examples

- “Base SDE Models” on page 14-14
- “Drift and Diffusion Models” on page 14-16
- “Linear Drift Models” on page 14-19
- “Parametric Models” on page 14-21

More About

- “SDEs” on page 14-2
- “SDE Class Hierarchy” on page 14-5

Base SDE Models

In this section...

Overview

The base sde object

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

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.

Creating an sde object using sde 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

Create an sde object using sde to represent a univariate geometric Brownian Motion model of the form:

$$dX_t = 0.1X_tdt + 0.3X_tdW_t$$

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 sde to create an sde object:

```
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
```

The `sde` object 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

`sde` | `bm` | `gbm` | `merton` | `bates` | `drift` | `diffusion` | `sdeddo` | `sdeld` | `cev` | `cir` | `heston` | `hwv` | `sdemrd` | `ts2func` | `simulate` | `simByEuler` | `simBySolution` | `simByQuadExp` | `simBySolution` | `interpolate`

Related Examples

- “Drift and Diffusion Models” on page 14-16
- “Linear Drift Models” on page 14-19
- “Parametric Models” on page 14-21

More About

- “SDEs” on page 14-2
- “SDE Models” on page 14-7
- “SDE Class Hierarchy” on page 14-5

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 objects 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` object 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` object 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` objects in the most general form to emphasize the functional (t, X_t) interface. However, you can specify the components `A` and `B` as functions that adhere to the common (t, X_t) interface, or as MATLAB arrays of appropriate dimension.

Example: Drift and Diffusion Rates

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

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` objects 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` object derives from the base`sde` object. To use this object, you must pass drift and diffusion-rate objects to `sdeddo`.

1 Create `drift` and `diffusion` rate objects:

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

2 Pass these objects to the `sdeddo` object:

```
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

`sde` | `bm` | `gbm` | `merton` | `bates` | `drift` | `diffusion` | `sdeddo` | `sdeld` | `cev` | `cir` | `heston` | `hwv` | `sdemrd` | `ts2func` | `simulate` | `simByEuler` | `simBySolution` | `simBySolution` | `interpolate` | `simByQuadExp`

Related Examples

- “Base SDE Models” on page 14-14
- “Linear Drift Models” on page 14-19
- “Parametric Models” on page 14-21

More About

- “SDEs” on page 14-2
- “SDE Models” on page 14-7
- “SDE Class Hierarchy” on page 14-5

Linear Drift Models

In this section...

Overview

The `sdeld` class derives from the `sdeddo` class. The `sdeld` 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 14-23. 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 14-14 using `sdeld`:

```
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

`sde` | `bm` | `gbm` | `merton` | `bates` | `drift` | `diffusion` | `sdeddo` | `sdeld` | `cev` | `cir` | `heston` | `hwv` | `sdemrd` | `ts2func` | `simulate` | `simByQuadExp` | `simByEuler` | `simBySolution` | `simBySolution` | `interpolate`

Related Examples

- “Base SDE Models” on page 14-14
- “Drift and Diffusion Models” on page 14-16
- “Parametric Models” on page 14-21

More About

- “SDEs” on page 14-2
- “SDE Models” on page 14-7
- “SDE Class Hierarchy” on page 14-5

Parametric Models

In this section...

Creating Brownian Motion (BM) Models

The Brownian Motion (BM) model (`bm`) derives directly from the linear drift (`sdeld`) model:

$$dX_t = \mu(t)dt + V(t)dW_t$$

Example: BM Models

Create a univariate Brownian motion (`bm`) object to represent the model using `bm`:

$$dX_t = 0.3dW_t .$$

```
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` object 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`) model:

$$dX_t = \mu(t)X_t dt + D(t, X_t^{\alpha(t)})V(t)dW_t$$

The `cev` object 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 `cev`:

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

```
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` object 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 `gbm`:

$$dX_t = 0.25X_tdt + 0.3X_tdW_t$$

```
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` object derives directly from the `sdeddo` object. 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 `sdemrd` 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 object, 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 cir to represent the same model as in “Example: SDEMRD Models” on page 14-23:

```
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 object, 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 hwv 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 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 object 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 object encompasses the models.

Creating Heston Stochastic Volatility Models

The Heston (heston) object derives directly from SDE from the Drift and Diffusion (sdeddo) class. Each Heston model is a bivariate composite model, consisting of two coupled univariate models:

$$dX_{1t} = B(t)X_{1t}dt + \sqrt{X_{2t}}X_{1t}dW_{1t} \tag{14-5}$$

$$dX_{2t} = S(t)[L(t) - X_{2t}]dt + V(t)\sqrt{X_{2t}}dW_{2t} \tag{14-6}$$

“Equation 14-5” is typically associated with a price process. “Equation 14-6” represents the evolution of the price process' variance. Models of type heston are typically used to price equity options.

Example: Heston Models

Create a heston object using heston 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

sde | bm | gbm | merton | bates | drift | diffusion | sdeddo | sdeld | cev | cir | heston | hwv | sdemrd | ts2func | simulate | simByEuler | simByQuadExp | simBySolution | simBySolution | interpolate

Related Examples

- “Base SDE Models” on page 14-14
- “Drift and Diffusion Models” on page 14-16
- “Linear Drift Models” on page 14-19

More About

- “SDEs” on page 14-2
- “SDE Models” on page 14-7
- “SDE Class Hierarchy” on page 14-5

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:

$$dX_t = \mu X_t dt + D(X_t)\sigma dW_t \tag{14-7}$$

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 `sde` 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 `sde` to create an `sde` object to represent the market model in "Equation 14-7":

```
SDE = sde(F, G, 'Correlation', correlation, 'StartState', X)
```

```
SDE =
   Class SDE: Stochastic Differential Equation
   -------------------------------------------
     Dimensions: State = 6, Brownian = 6
   -------------------------------------------
      StartTime: 0
     StartState: 100 (6x1 double array)
    Correlation: 6x6 double array
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
```

The `sde` object requires additional information to determine the dimensionality of the model, because the functions passed to the`sde` object are known only by their (t, X_t) interface. In other words, the`sde` object 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 `sdeddo` to represent the market model in "Equation 14-7":

1. Load the `Data_GlobalIdx2` data set:

   ```
   load Data_GlobalIdx2
   prices  = [Dataset.TSX Dataset.CAC Dataset.DAX ...
   Dataset.NIK Dataset.FTSE Dataset.SP];
   ```
2. Convert daily prices to returns:

   ```
   returns =  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 `drift` and `diffusion`:

   ```
   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 `sdeddo`:

```
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` object requires two input objects that provide more information than the two functions from step 4 of “Representing Market Models Using SDE Objects” on page 14-28. 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` object, 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 14-7”.

1 Load the `Data_GlobalIdx2` data set:

```
load Data_GlobalIdx2
prices  = [Dataset.TSX Dataset.CAC Dataset.DAX ...
    Dataset.NIK Dataset.FTSE Dataset.SP];
```

2 Convert daily prices to returns:

```
returns =  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 `sdeld`:

```
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 `cev`:

```
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 `gbm`:

```
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` object 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 `sde` to specify the SDE model as in “Representing Market Models Using SDE Objects” on page 14-28.

```
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);
```

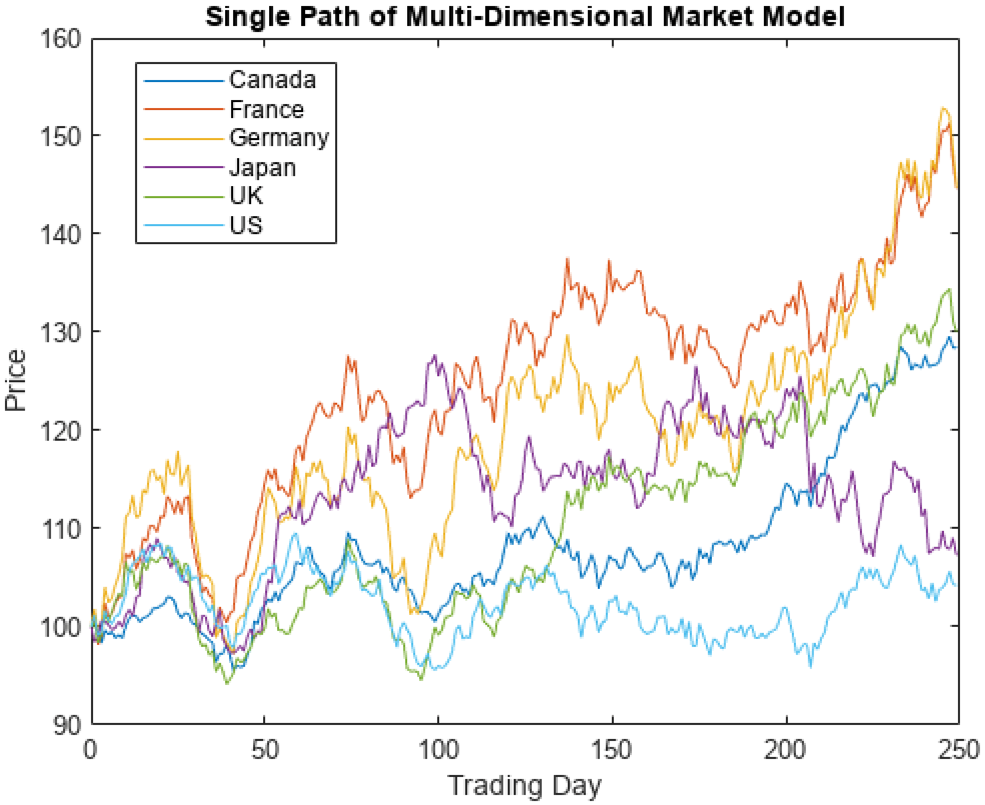
2 Simulate a single path of correlated equity index prices over one calendar year (defined as approximately 250 trading days) using the default`simulate` method:

```
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 14-32:

1 Load the `Data_GlobalIdx2` data set and use `sde` to specify the SDE model as in “Representing Market Models Using SDE Objects” on page 14-28.

```
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:

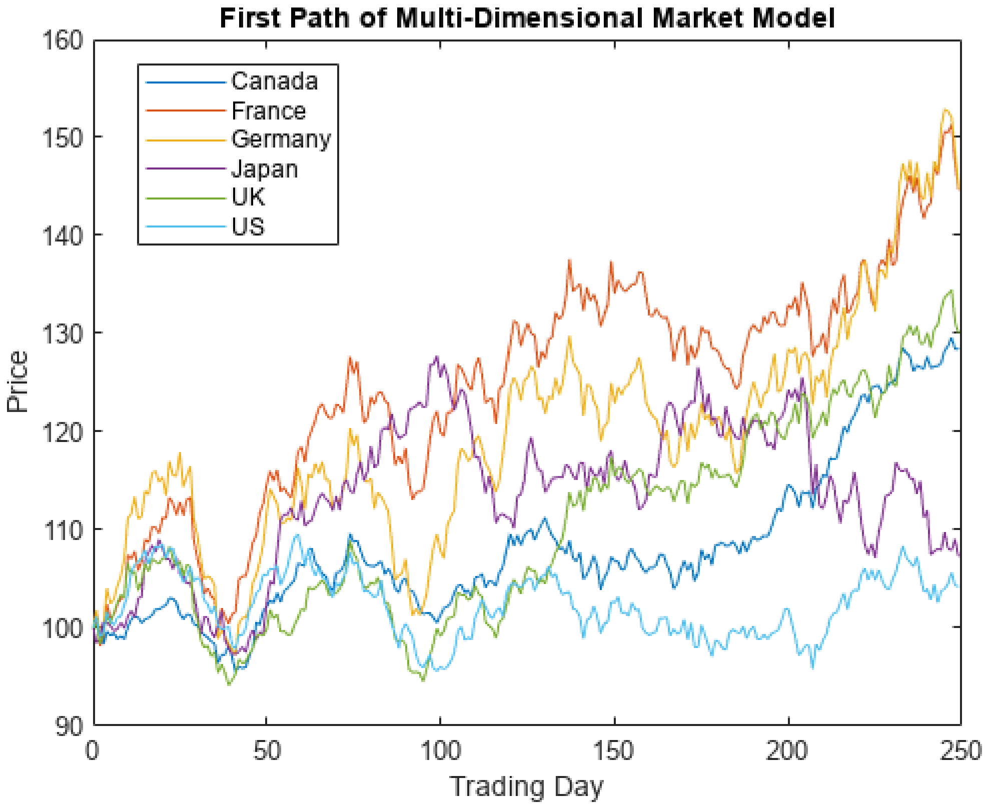
```
rng(142857,'twister')
[S,T] = simulate(SDE, nPeriods, 'DeltaTime', dt, 'nTrials', 10);

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 function, simBySolution, provides an approximate solution of the underlying stochastic differential equation, designed for accuracy

1 Load the Data_GlobalIdx2 data set and use sde to specify the SDE model as in “Representing Market Models Using SDE Objects” on page 14-28, and the GBM model as in “Representing Market Models Using SDELD, CEV, and GBM Objects” on page 14-30.

```
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.

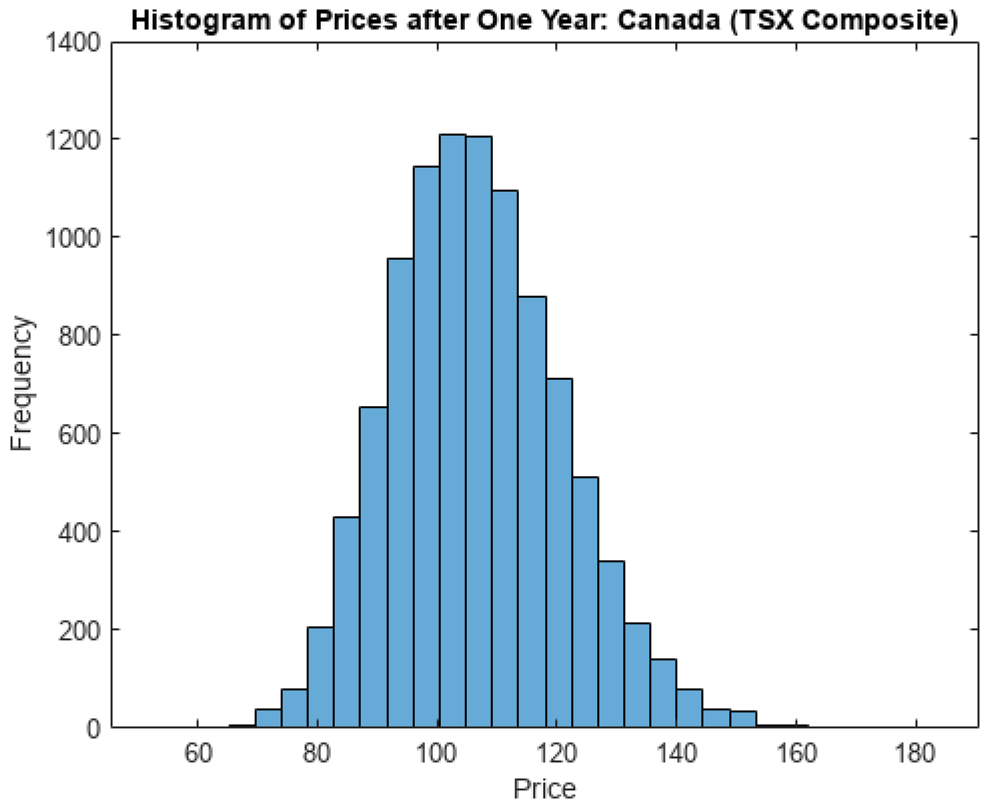
```
nPeriods = 249;      % # of simulated observations
dt       =   1;      % time increment = 1 day
rng(142857,'twister')
[X,T] = simulate(GBM, nPeriods, 'DeltaTime', dt, ...
    'nTrials', 10000);

whos X
```

```
  Name        Size                     Bytes  Class     Attributes

  X         250x6x10000            120000000  double
```

The output X is a much larger time series array.

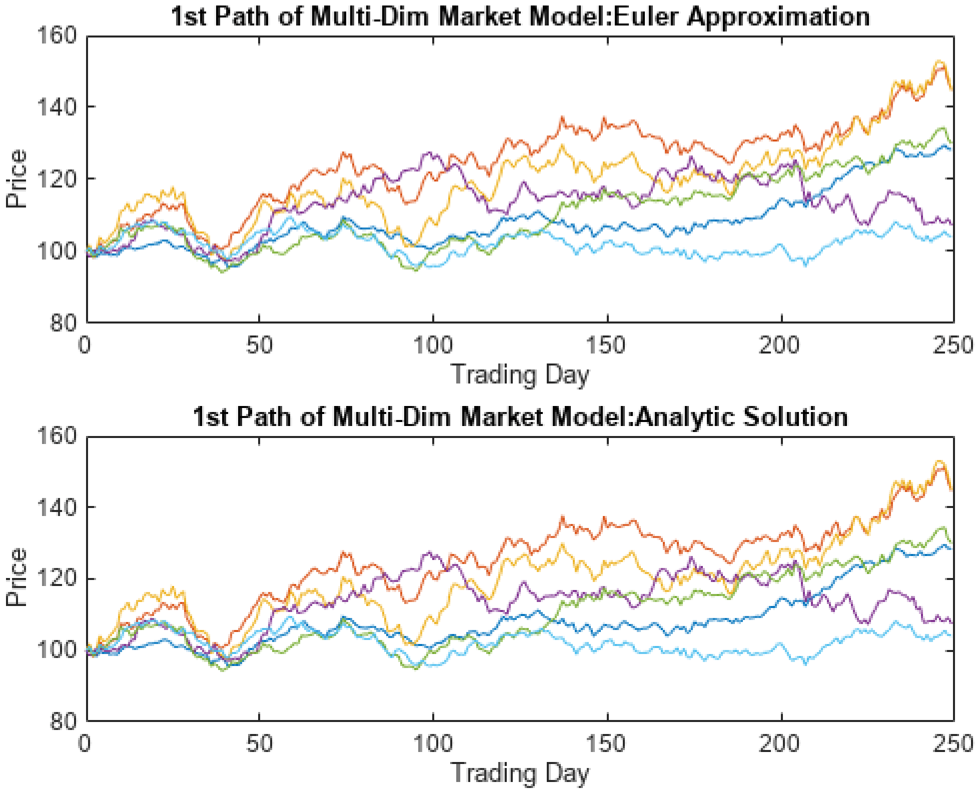
3 Using this sample size, examine the terminal distribution of Canada's TSX Composite to verify qualitatively the lognormal character of the data.

```
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:

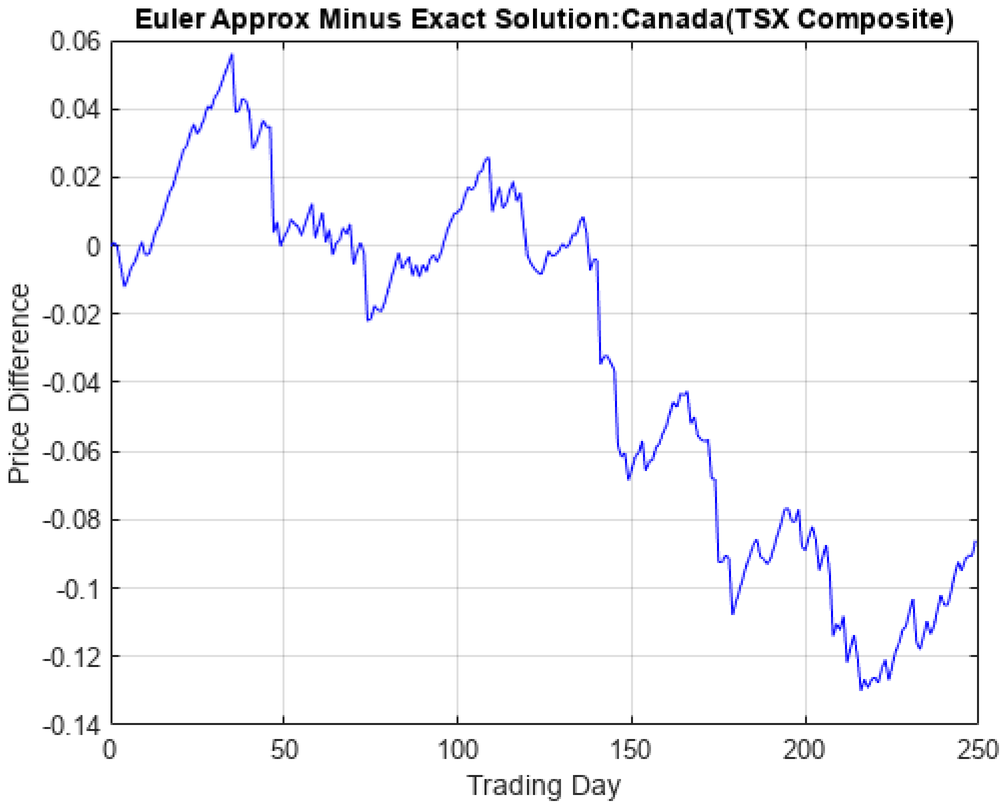
```
rng(142857,'twister')
[S,T] = simulate(SDE, nPeriods, 'DeltaTime', dt, 'nTrials', 10);
rng(142857,'twister')
[X,T] = simBySolution(GBM, nPeriods,...
    'DeltaTime', dt, 'nTrials', 10);
subplot(2,1,1)
plot(T, S(:,:,1)), xlabel('Trading Day'),ylabel('Price')
title('1st Path of Multi-Dim Market Model:Euler 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 is 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 14-65.

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 14-35). 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);
   ```

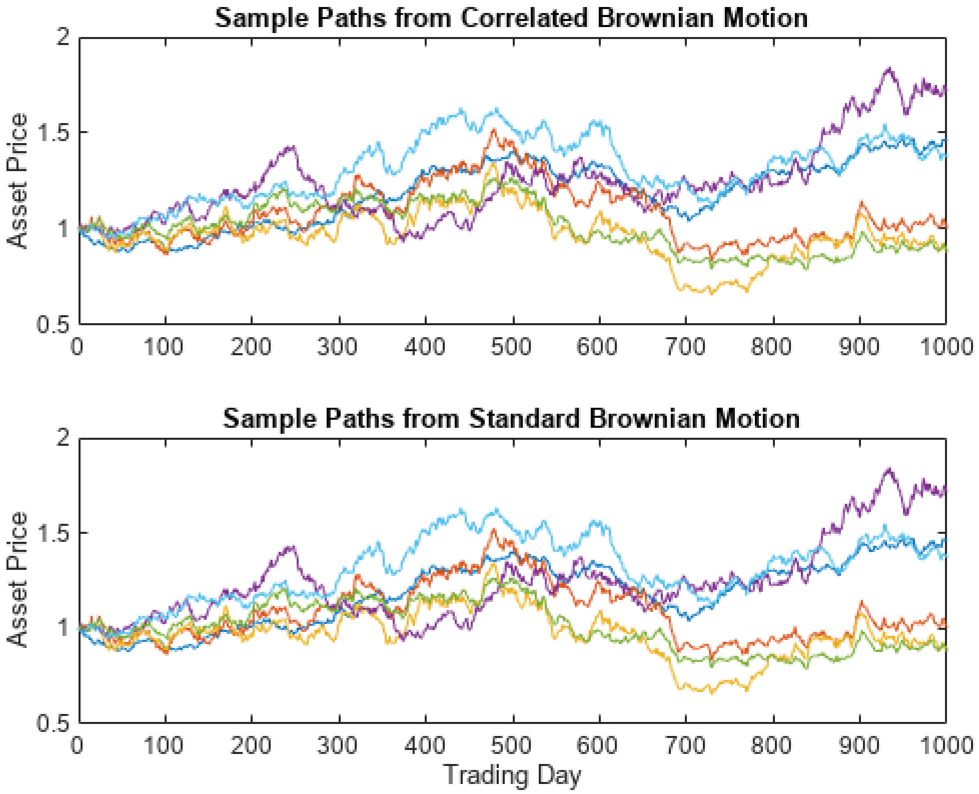
   Here, `sigma` captures both the correlation and magnitude of the asset return uncertainty. In contrast to the first technique, the components of the Brownian motion are independent. Also, this technique accepts the default assignment of an identity matrix to `Correlation`, and is more straightforward.

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')
```



Sample Paths from Correlated Brownian Motion

Sample Paths from Standard Brownian Motion

Dynamic Behavior of Market Parameters

As discussed in “Creating SDE Objects” on page 14-7, object parameters may be evaluated as if they are MATLAB functions accessible by a common interface. This accessibility provides the impression of dynamic behavior regardless of whether the underlying parameters are truly time-varying.

Furthermore, because parameters are accessible by a common interface, seemingly simple linear constructs may in fact represent complex, nonlinear designs.

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

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

In this model, the return, *μ(t)*, and volatility, *σ(t)*, are 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 three-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 14-65.

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:

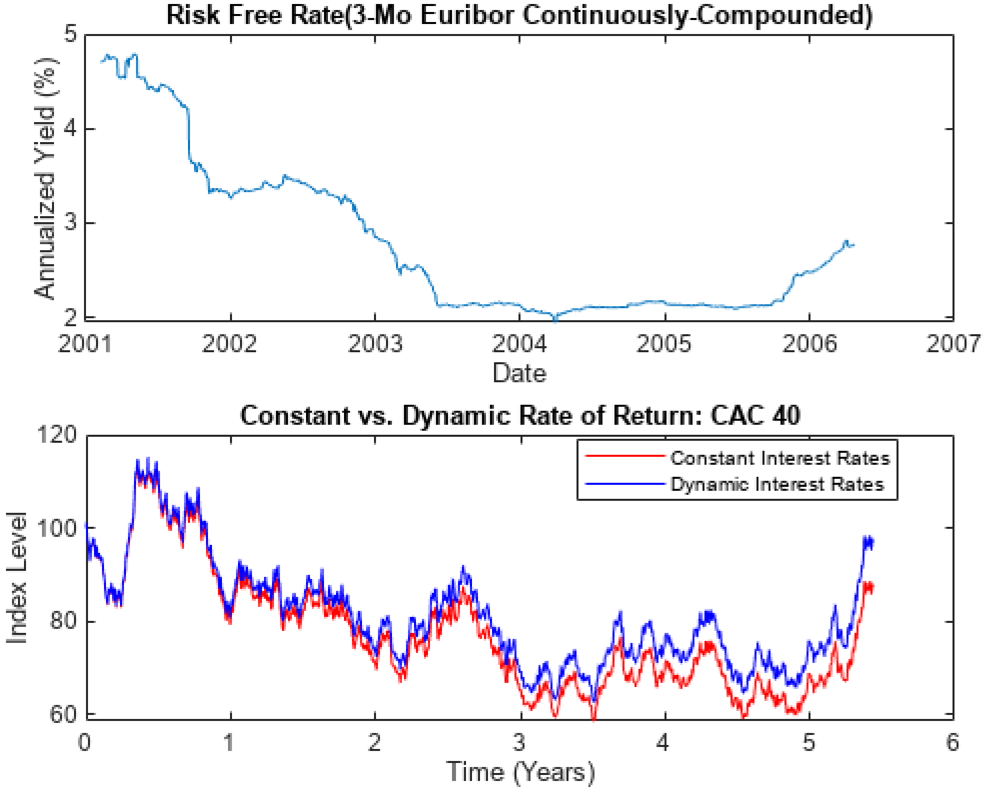
```
rng(5713,'twister')
obj = gbm(r, diag(sigma),'StartState',100)
```

```
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 14-53, 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:

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

See Also

`sde` | `bm` | `gbm` | `merton` | `bates` | `drift` | `diffusion` | `sdeddo` | `sdeld` | `cev` | `cir` | `heston` | `hwv` | `sdemrd` | `ts2func` | `simulate` | `simByEuler` | `simBySolution` | `simByQuadExp` | `simBySolution` | `interpolate`

Related Examples

- "Simulating Interest Rates" on page 14-48
- "Stratified Sampling" on page 14-57
- "Price American Basket Options Using Standard Monte Carlo and Quasi-Monte Carlo Simulation" on page 14-70
- "Improving Performance of Monte Carlo Simulation with Parallel Computing" on page 14-87
- "Base SDE Models" on page 14-14
- "Drift and Diffusion Models" on page 14-16
- "Linear Drift Models" on page 14-19
- "Parametric Models" on page 14-21

More About

- "SDEs" on page 14-2
- "SDE Models" on page 14-7
- "SDE Class Hierarchy" on page 14-5
- "Performance Considerations" on page 14-64

Simulating Interest Rates

In this section...

Simulating Interest Rates Using Interpolation

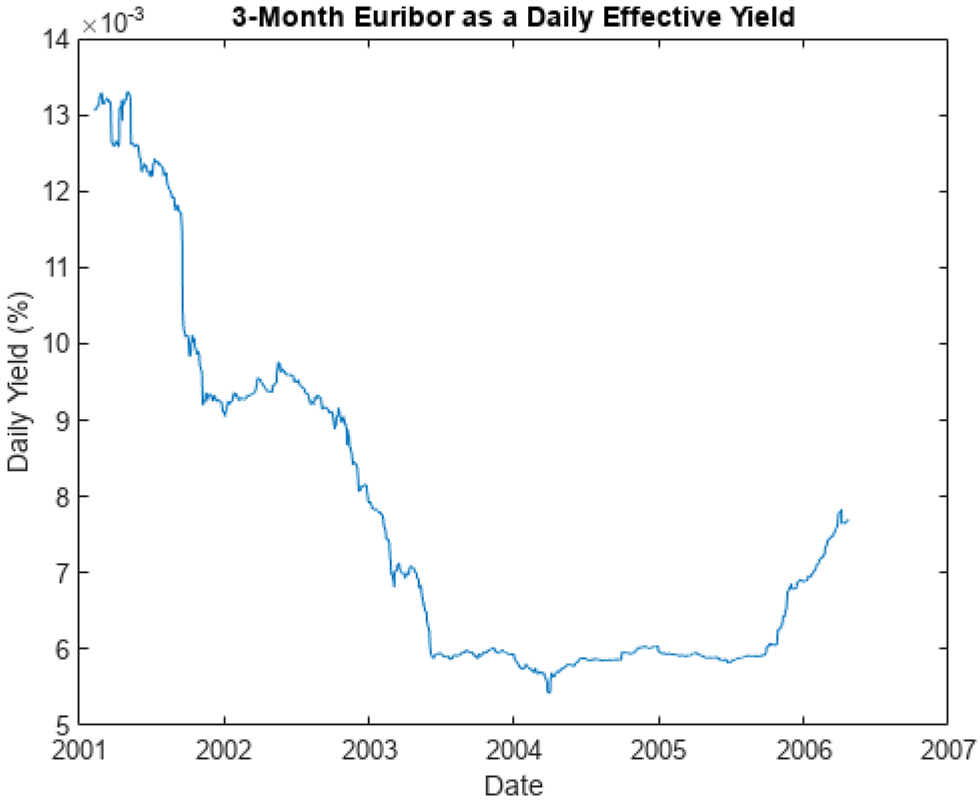
All simulation methods require that you specify a time grid by specifying the number of periods (NPeriods). You can also optionally specify a scalar or vector of strictly positive time increments (DeltaTime) and intermediate time steps (NSteps). These parameters, along with an initial sample time associated 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 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
```

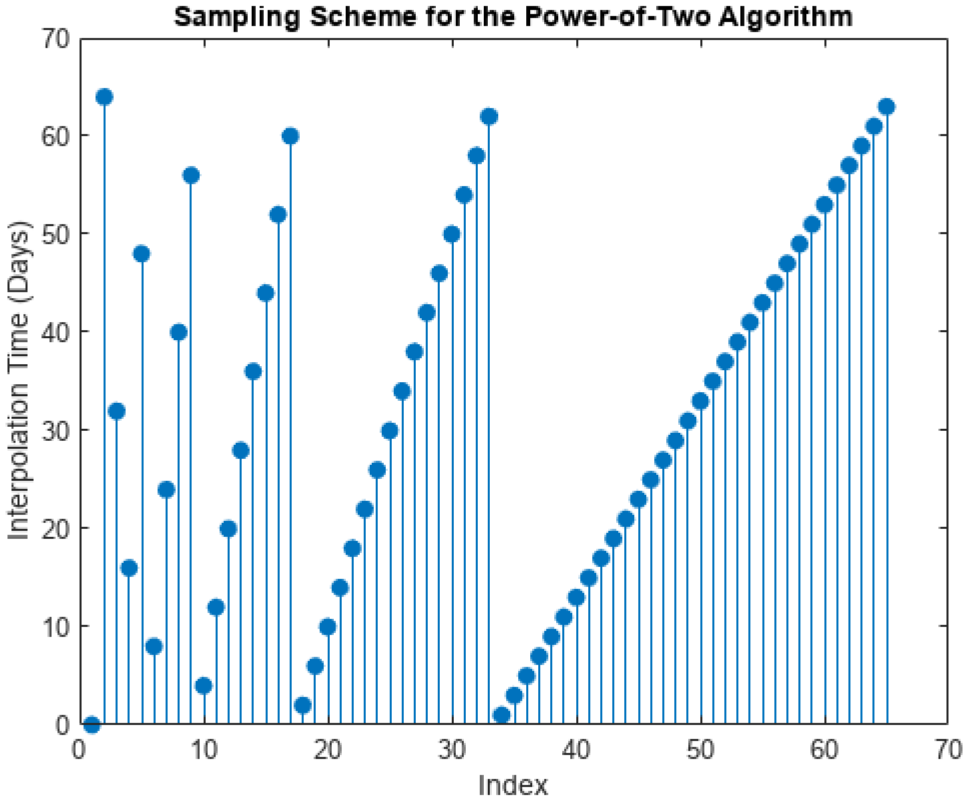
4 **Simulate the fitted model.** Assume, for example, that you simulate the fitted model over 64 (2^6) trading days, using a refined Brownian bridge with the power-of-two algorithm instead of the usual beginning-to-end Monte Carlo simulation approach. Furthermore, assume that the initial time and state 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:

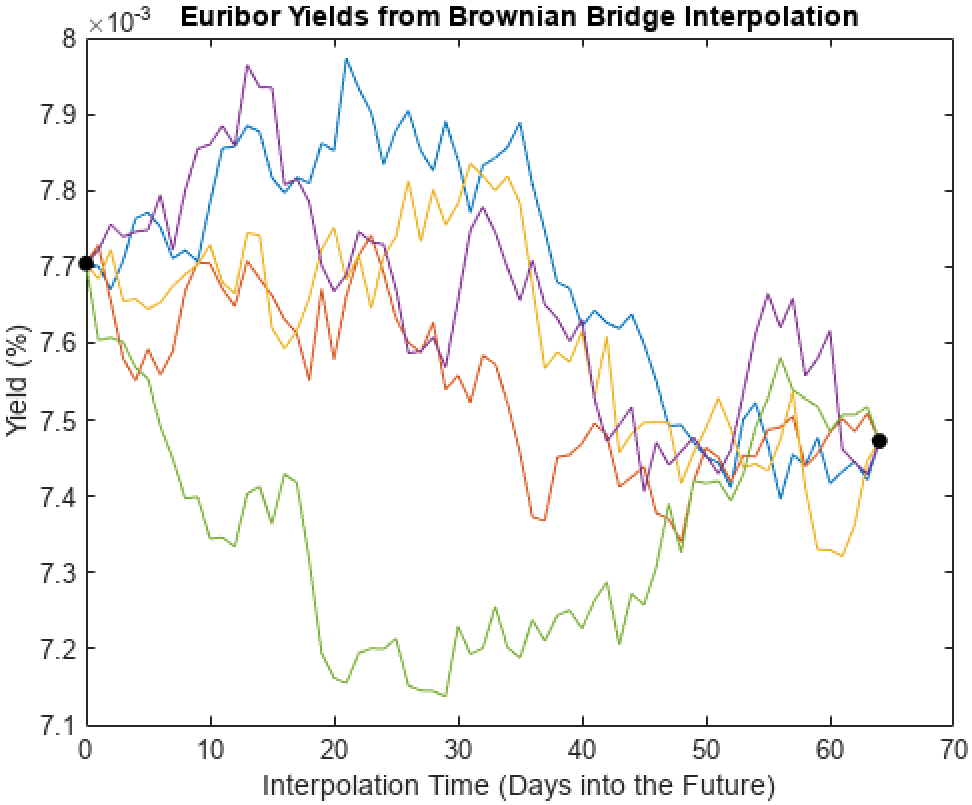
```
average = obj.StartState * exp(-speed * T) + level * ...
(1 - exp(-speed * T));
X       = [obj.StartState ; average];
```

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 14-57.

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 14-57.

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 14-65. 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 `cir`, 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
     '
```

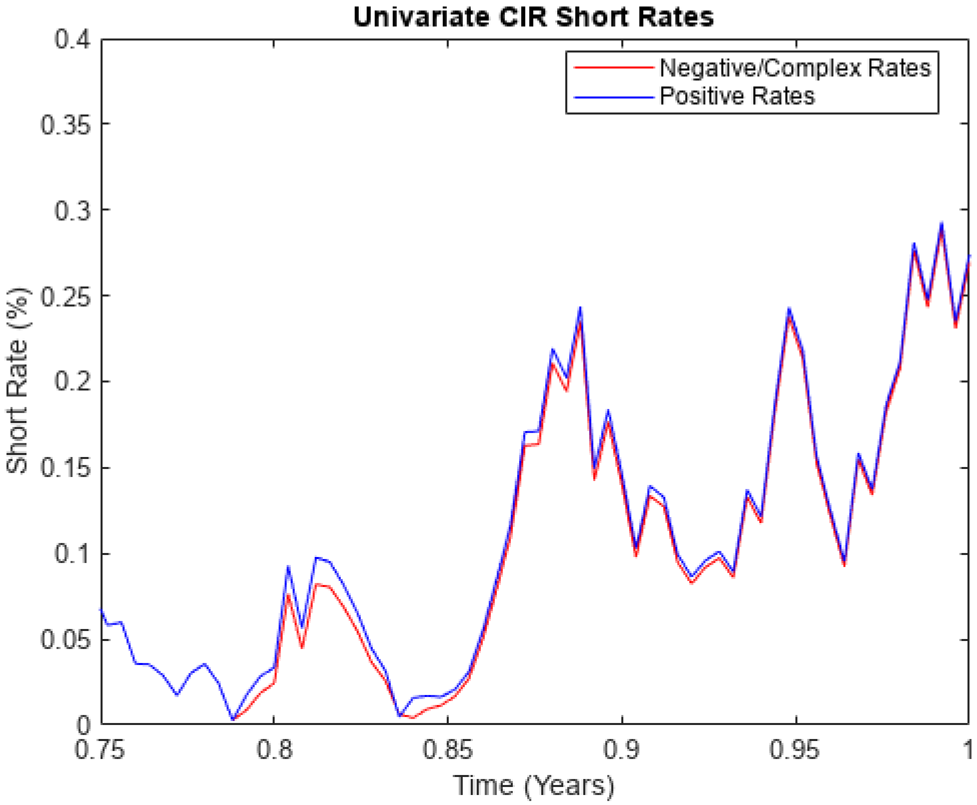
Interest rates can become negative if the resulting paths are simulated in discrete time. Moreover, since `cir` models incorporate a square root diffusion term, the short rates might even become complex.

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 14-64.

See Also

`sde` | `bm` | `gbm` | `merton` | `bates` | `drift` | `diffusion` | `sdeddo` | `sdeld` | `cev` | `cir` | `heston` | `hwv` | `sdemrd` | `ts2func` | `simulate` | `simByEuler` | `simByQuadExp` | `simBySolution` | `simBySolution` | `interpolate`

Related Examples

- "Simulating Equity Prices" on page 14-28
- "Stratified Sampling" on page 14-57
- "Price American Basket Options Using Standard Monte Carlo and Quasi-Monte Carlo Simulation" on page 14-70
- "Improving Performance of Monte Carlo Simulation with Parallel Computing" on page 14-87
- "Base SDE Models" on page 14-14
- "Drift and Diffusion Models" on page 14-16
- "Linear Drift Models" on page 14-19
- "Parametric Models" on page 14-21

More About

- “SDEs” on page 14-2
- “SDE Models” on page 14-7
- “SDE Class Hierarchy” on page 14-5
- “Performance Considerations” on page 14-64

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 14-48. 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 `Example_StratifiedRNG.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:

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 two-dimensional equivalent array:

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

4 Verify the stratification:

   a Recreate the uniform draws with proportional sampling:

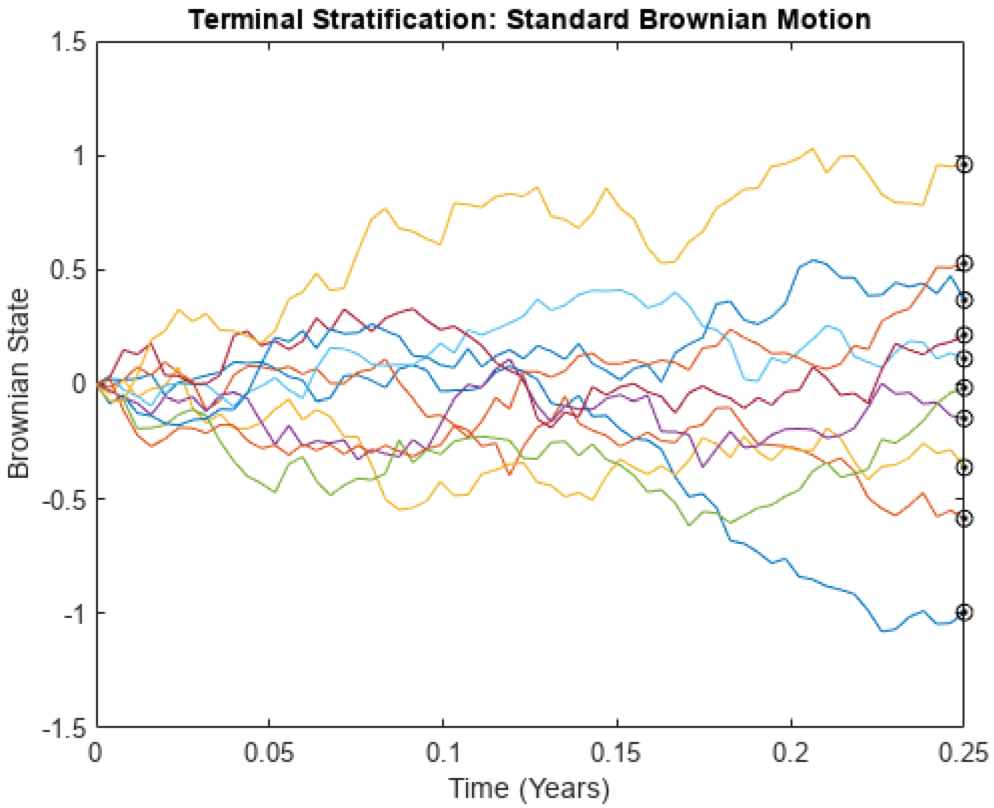
   ```
   rng(10203,'twister')
   U  = ((1:nPaths)' - 1 + rand(nPaths,1))/nPaths;
   ```

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:

$$dX_t = rX_tdt + \sigma X_tdW_t$$

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));
```

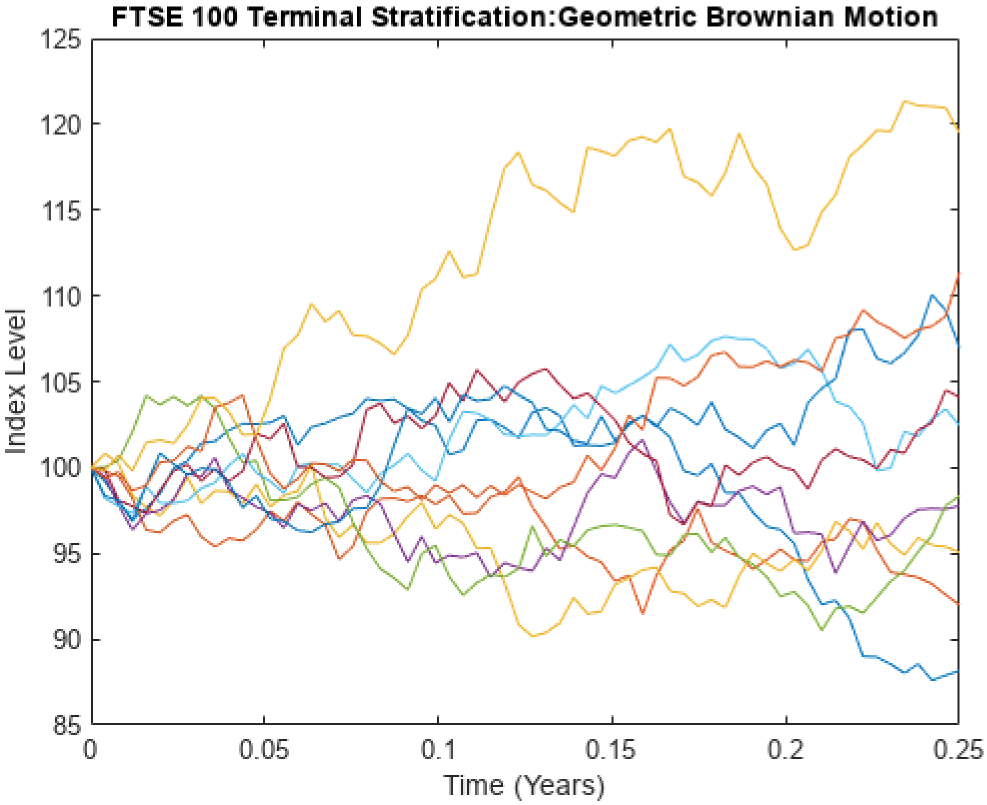
2 Create the GBM model using gbm, assuming the FTSE 100 starts at 100:

```
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 Using Interpolation" on page 14-48.

See Also

`sde` | `bm` | `gbm` | `merton` | `bates` | `drift` | `diffusion` | `sdeddo` | `sdeld` | `cev` | `cir` | `heston` | `hwv` | `sdemrd` | `ts2func` | `simulate` | `simByEuler` | `simBySolution` | `simByQuadExp` | `simBySolution` | `interpolate`

Related Examples

- "Simulating Equity Prices" on page 14-28
- "Simulating Interest Rates" on page 14-48
- "Price American Basket Options Using Standard Monte Carlo and Quasi-Monte Carlo Simulation" on page 14-70
- "Improving Performance of Monte Carlo Simulation with Parallel Computing" on page 14-87
- "Base SDE Models" on page 14-14
- "Drift and Diffusion Models" on page 14-16

- “Linear Drift Models” on page 14-19
- “Parametric Models” on page 14-21

More About

- “SDEs” on page 14-2
- “SDE Models” on page 14-7
- “SDE Class Hierarchy” on page 14-5
- “Performance Considerations” on page 14-64

Quasi-Monte Carlo Simulation

Quasi-Monte Carlo simulation is a Monte Carlo simulation that uses quasi-random sequences instead of pseudo random numbers. The quasi-random sequences, also called low-discrepancy sequences, are deterministic, uniformly distributed sequences that are specifically designed to place sample points as uniformly as possible. In many cases, this distributed sequences improves the performance of Monte Carlo simulations with faster computational times and sometimes higher accuracy.

The standard Monte Carlo simulation using pseudo random numbers has a convergence rate of only $O(N^{-1/2})$, while the quasi-Monte Carlo rate of convergence can be much faster with an error of $O(N^{-1})$ in the best cases. For example, for a standard Monte Carlo simulation, it is necessary to increase 100 times the number of simulations *NTrials* to reduce the error by a factor of 10, whereas a quasi-Monte Carlo simulation requires less, or much less, than 100 times to achieve the same goal.

Quasi-Monte Carlo simulation produces a purely deterministic result. Therefore, when computing the variance and constructing a confidence band for the estimates, randomized quasi-Monte Carlo simulation is useful because of faster computational times and sometimes higher accuracy. You can also use randomized quasi-Monte Carlo to introduce randomization into the low-discrepancy sequences.

The starting point for a Monte Carlo simulation is the construction of a Brownian motion sample path (or Wiener path). Such paths are built from a set of independent Gaussian variates, using either standard discretization, Brownian-bridge construction, or principal components construction.

To perform Quasi-Monte Carlo simulation, the name-value arguments for `MonteCarloMethod`, `QuasiSequence`, and `BrownianMotionMethod` by using the supported methods for the following SDE objects. For example, `[paths,time,z] = simByEuler(cir_obj,10,'ntrials',4096,'method','basic','montecarlomethod','quasi','quasisequence','sobol','brownianmotionmethod','principal-components')` performs a Quasi-Monte Carlo simulation using a principal components construction method with a `cir` model.

Class Names	**Methods Supporting Quasi-Monte Carlo Simulation and Brownian Bridge Construction**
`sde`	`simByEuler`, `simulate`
`bm`	`simByEuler`, `simulate`
`gbm`	`simByEuler`, `simBySolution`, `simulate`
`merton`	`simBySolution`, `simByMilstein`, `simulate`
`bates`	`simByTransition`, `simByQuadExp`, `simByMilstein`, `simulate`
`hwv`	`simBySolution`, `simulate`
`heston`	`simByTransition`, `simByQuadExp`, `simByMilstein`, `simulate`
`cev`	`simByEuler`, `simulate`
`cir`	`simByEuler`, `simByTransition`, `simByQuadExp`, `simByMilstein`, `simulate`
`sdeddo`	`simByEuler`, `simulate`

Class Names	Methods Supporting Quasi-Monte Carlo Simulation and Brownian Bridge Construction
sdeld	simByEuler, simulate
sdemrd	simByEuler, simulate

See Also

sde | bm | gbm | merton | bates | drift | diffusion | sdeddo | sdeld | cev | cir | heston | hwv | sdemrd | ts2func | simulate | simByEuler | simBySolution | simByQuadExp | simBySolution | simByMilstein | simByMilstein | simByMilstein | simByMilstein | interpolate

Related Examples

- “Simulating Equity Prices” on page 14-28
- “Simulating Interest Rates” on page 14-48
- “Price American Basket Options Using Standard Monte Carlo and Quasi-Monte Carlo Simulation” on page 14-70
- “Improving Performance of Monte Carlo Simulation with Parallel Computing” on page 14-87
- “Base SDE Models” on page 14-14
- “Drift and Diffusion Models” on page 14-16
- “Linear Drift Models” on page 14-19
- “Parametric Models” on page 14-21

More About

- “SDEs” on page 14-2
- “SDE Models” on page 14-7
- “SDE Class Hierarchy” on page 14-5
- “Performance Considerations” on page 14-64

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 14-64
- "Managing Memory Using End-of-Period Processing Functions" on page 14-65

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 14-65).

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 14-45.

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 14-21 with constant parameters, simulated by Euler approximation (`simByEuler`).
- "Creating Geometric Brownian Motion (GBM) Models" on page 14-22 with constant parameters, simulated by closed-form solution (`simBySolution`).
- "Creating Hull-White/Vasicek (HWV) Gaussian Diffusion Models" on page 14-25 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 `gbm` with constant parameters:

$$dX_t = 0.1X_tdt + 0.4X_tdW_t\,.$$

Assume that the expected rate of return and volatility parameters are annualized, and that a calendar year comprises 250 trading days.

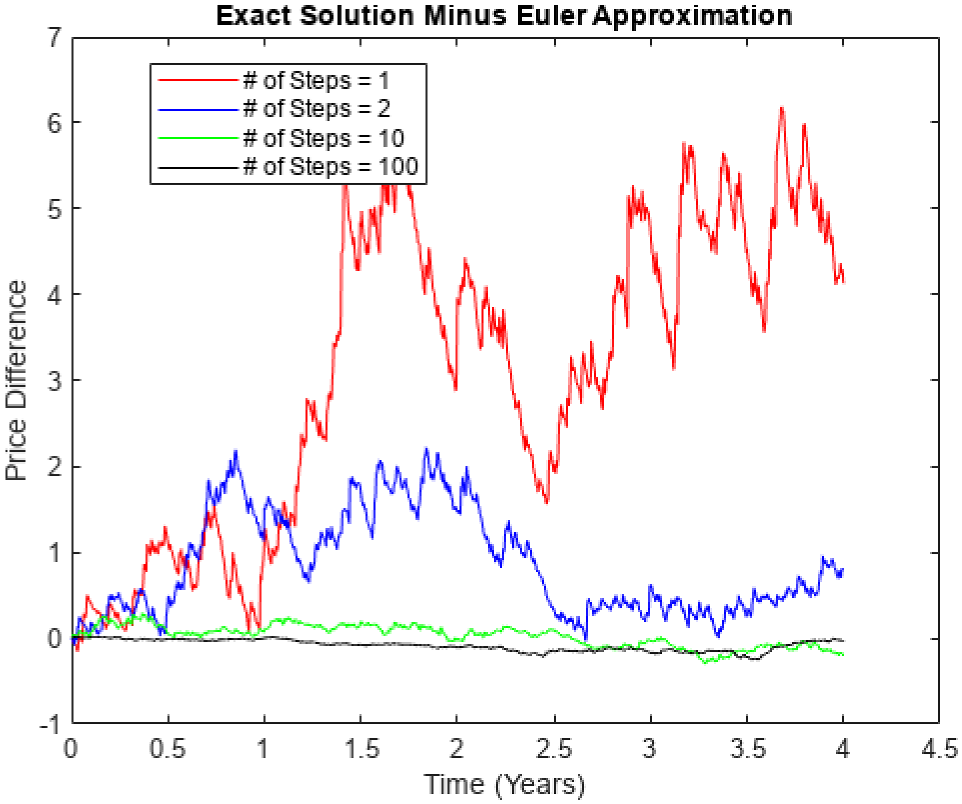
1 Simulate approximately four years of univariate prices for both the exact solution and the Euler approximation for various values of `NSteps`:

```
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
```



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 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

`sde` | `bm` | `gbm` | `merton` | `bates` | `drift` | `diffusion` | `sdeddo` | `sdeld` | `cev` | `cir` | `heston` | `hwv` | `sdemrd` | `ts2func` | `simulate` | `simByEuler` | `interpolate` | `simByQuadExp` | `simBySolution` | `simBySolution`

Related Examples

- “Simulating Equity Prices” on page 14-28
- “Simulating Interest Rates” on page 14-48
- “Price American Basket Options Using Standard Monte Carlo and Quasi-Monte Carlo Simulation” on page 14-70
- “Improving Performance of Monte Carlo Simulation with Parallel Computing” on page 14-87
- “Base SDE Models” on page 14-14
- “Drift and Diffusion Models” on page 14-16
- “Linear Drift Models” on page 14-19
- “Parametric Models” on page 14-21

More About

- “SDEs” on page 14-2
- “SDE Models” on page 14-7
- “SDE Class Hierarchy” on page 14-5

Price American Basket Options Using Standard Monte Carlo and Quasi-Monte Carlo Simulation

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)
- Quasi-Monte Carlo simulation using for a `gbm` object using `simByEuler`.

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:

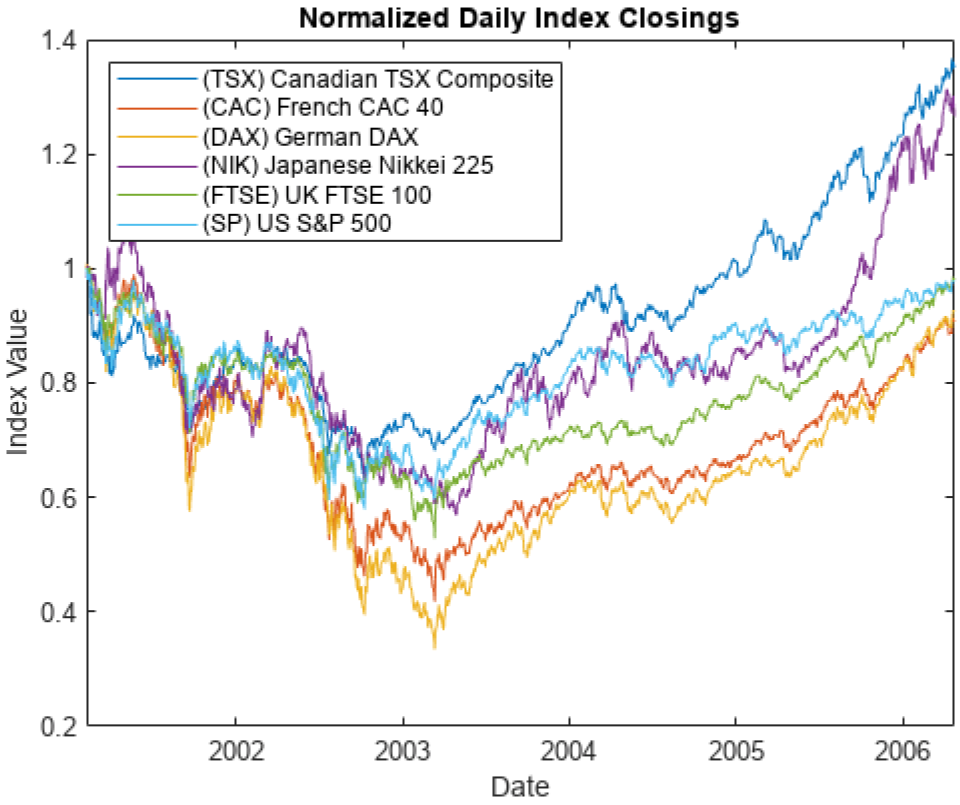
- TSX Composite (Canada)
- CAC 40 (France)
- DAX (Germany)
- Nikkei 225 (Japan)
- FTSE 100 (UK)
- S&P 500 (US)

```
clear
load Data_GlobalIdx2
dates = datetime(dates,'ConvertFrom','datenum');
```

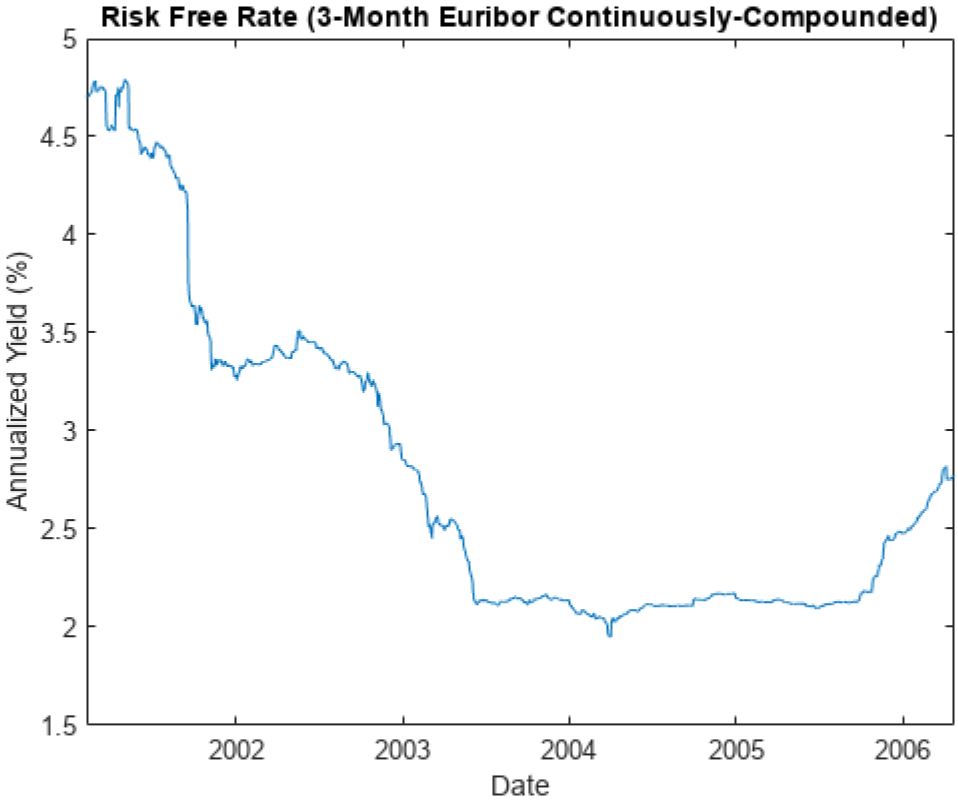
The following plots illustrate this data. Specifically, the plots show the relative price movements of each index and the Euribor risk-free rate proxy. The initial level of each index has been normalized to unity to facilitate the comparison of relative performance over the historical record.

```
nIndices  = size(Data,2)-1;     % Number 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,'Method','continuous'),'Method','continuous'))

xlabel('Date')
```

```
ylabel('Index Value')
title ('Normalized Daily Index Closings')
legend(series{1:end-1}, 'Location', 'NorthWest')
```



```
plot(dates, 100 * yields)
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.

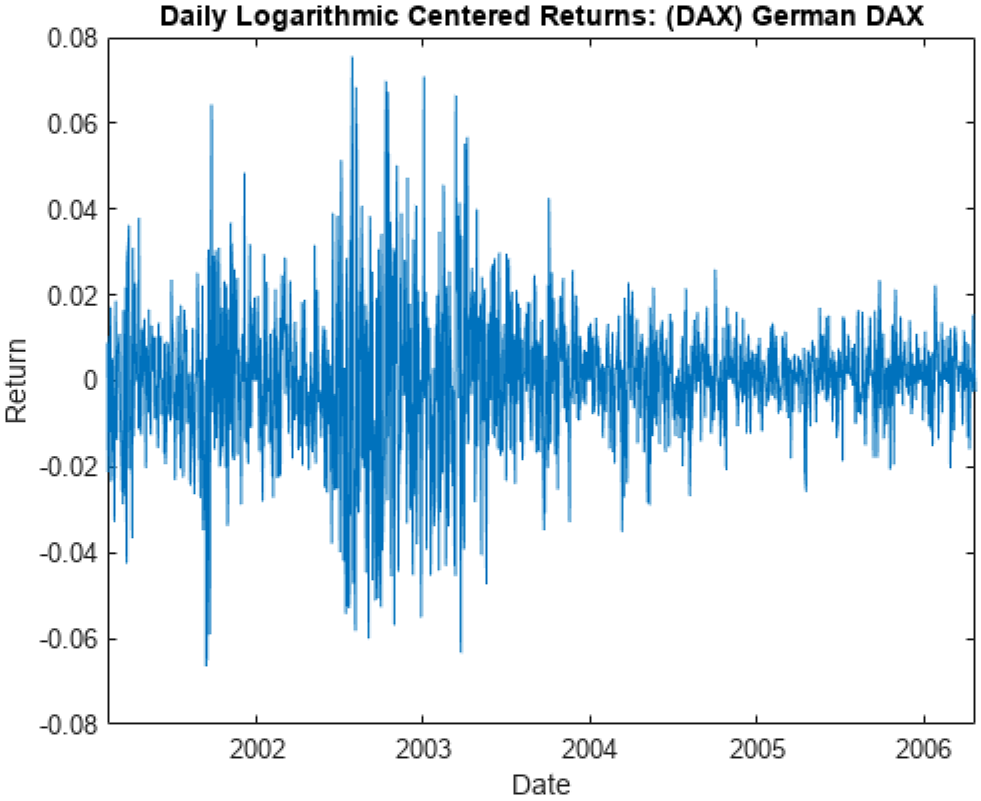
The following analysis highlights the Pareto distribution, which is more widely used in risk management applications.

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,'Method','continuous');  % Convert prices to returns
returns = returns - mean(returns);  % Center the returns
index   = 3;                        % Germany stored in column 3

plot(dates(2:end), returns(:,index))
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.

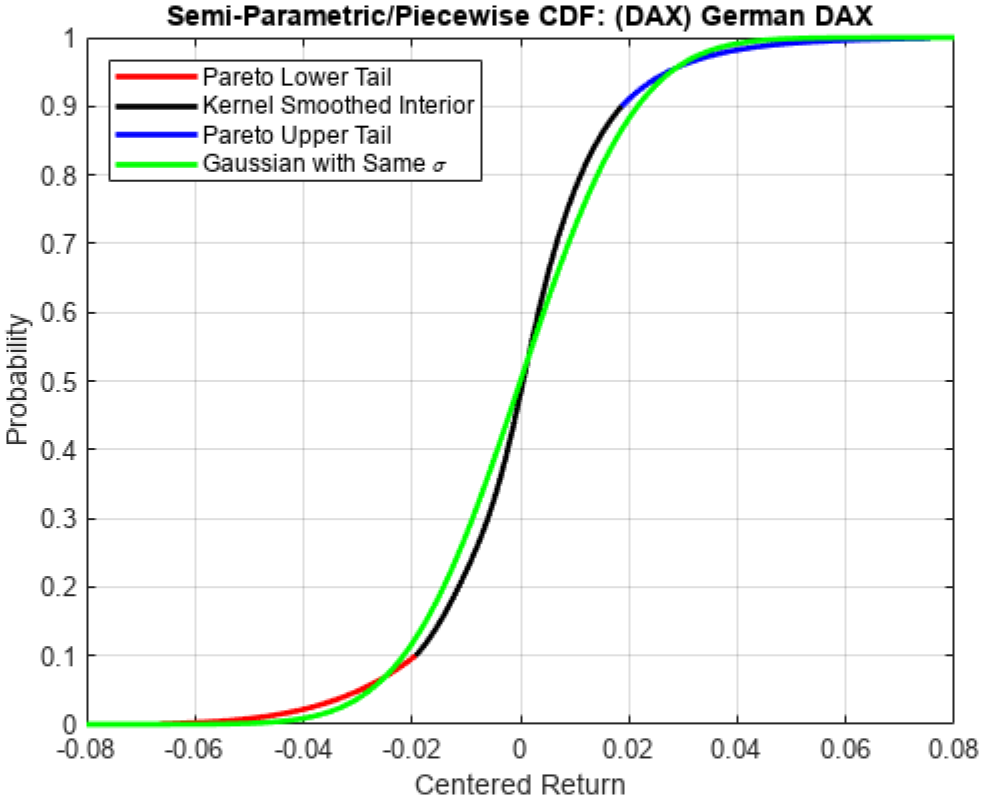
Now that three distinct regions of the piecewise distribution have been estimated, graphically concatenate and display the result.

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 = 6×6

    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 = 6×6

    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 = 6×6

    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
```

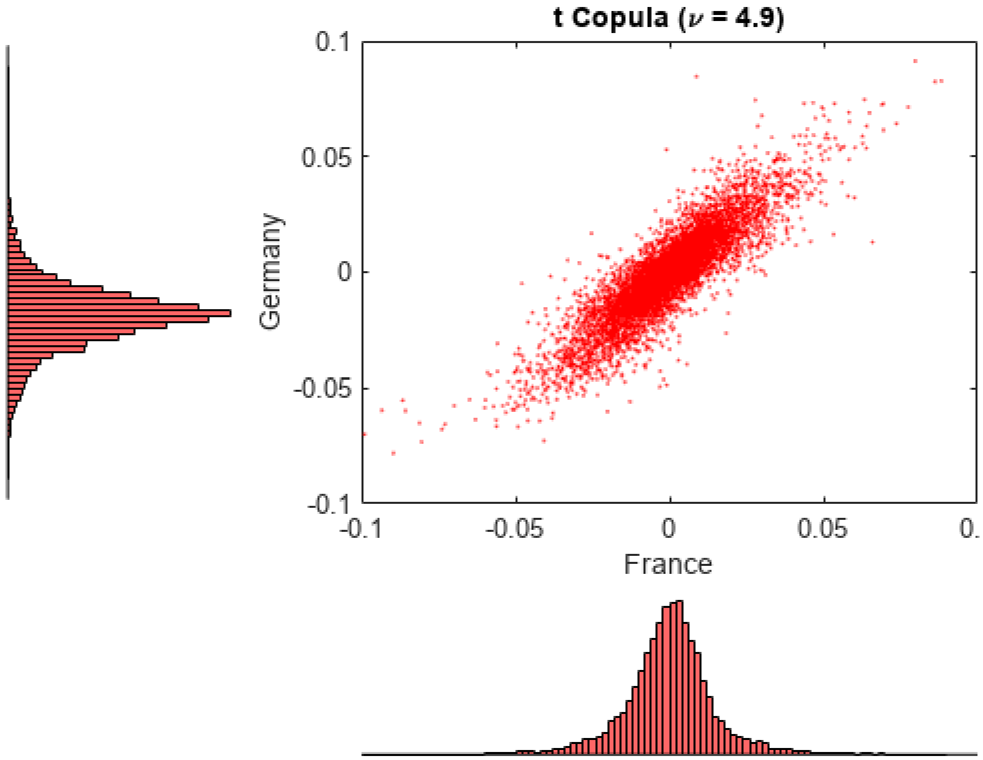
Copula Simulation

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.

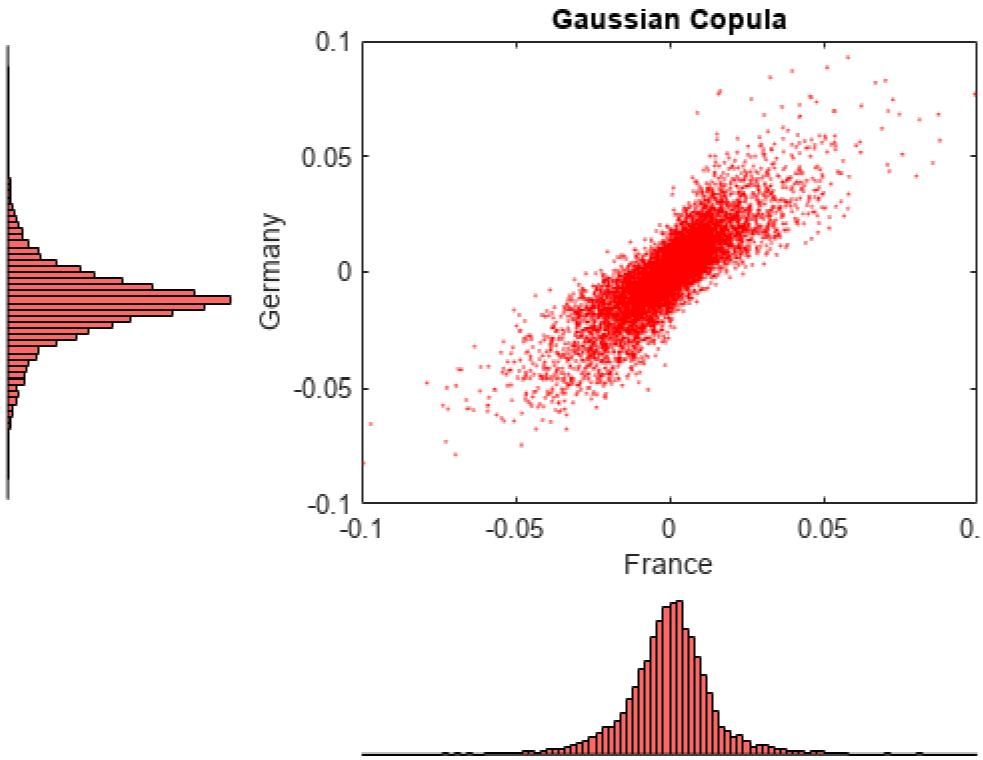
The following code segment illustrates the dependence structure by simulating centered returns using the *t* copula. It then plots a 2-D scatter plot with marginal histograms for the French CAC 40 and German DAX using the Statistics and Machine Learning Toolbox `scatterhist` function. The French and German indices were chosen simply because they have the highest correlation of the available data.

```
nPoints = 10000;                          % Number 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.

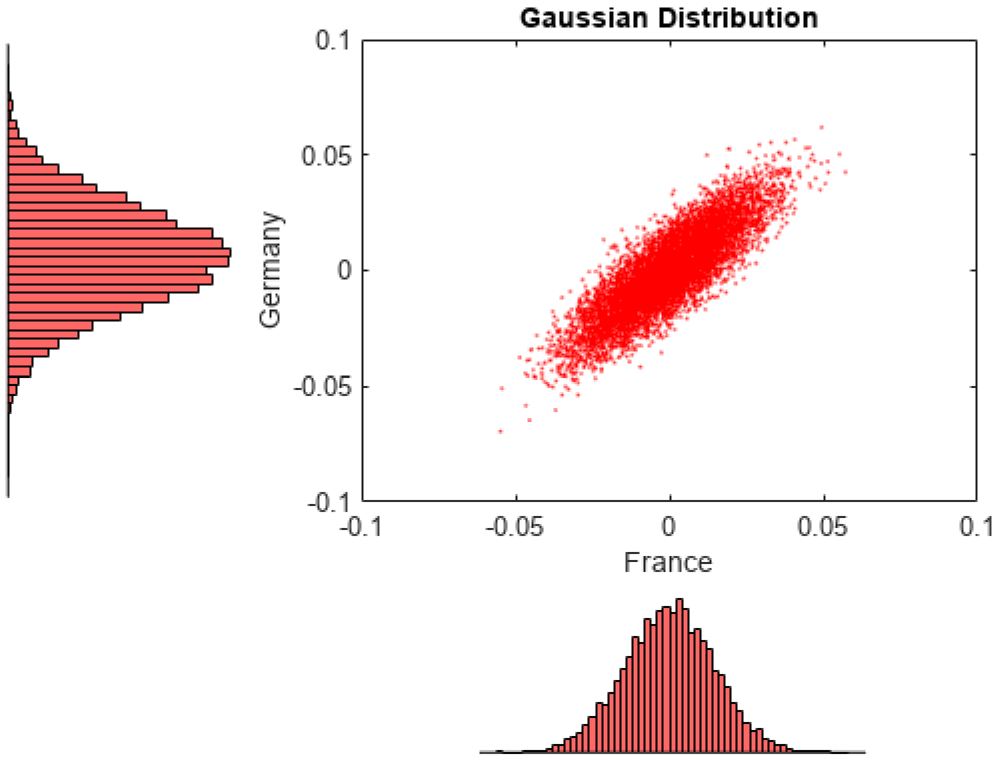
However, the scatter graph of each figure indicates the dependence structure associated with the given copula, and in contrast to the univariate margins shown in the histograms, the scatter graphs are distinct.

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;                      % Number of independent trials
nPeriods = 63;  % Number 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 “Inducing Dependence and Correlation” on page 14-40.

```
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 `simByEuler` method for both a standard Monte Carlo simulation and a Quasi-Monte Carlo simulation. 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)

f = Example_LongstaffSchwartz(nPeriods, nTrials);

simByEuler(GBM1, nPeriods, 'nTrials'  , nTrials, 'DeltaTime', dt, ...
                          'Processes', f.LongstaffSchwartz, ...
                          'MontecarloMethod','quasi');

QuasiMonteCarloCallPrice = f.CallPrice(strike, r);
QuasiMonteCarloPutPrice  = 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)
```

```
                    # of Monte Carlo Trials:      100
```

```
fprintf('                    # of Time Periods/Trial: %8d\n\n'  , nPeriods)
```

```
                    # of Time Periods/Trial:       63
```

```
fprintf(' Brownian Motion American Call Basket Price: %8.4f\n'  , BrownianMotionCallPrice)
```

```
 Brownian Motion American Call Basket Price:  25.9456
```

```
fprintf(' Brownian Motion American Put  Basket Price: %8.4f\n\n', BrownianMotionPutPrice)
```

```
 Brownian Motion American Put  Basket Price:  16.4132
```

```
fprintf(' Quasi-Monte Carlo American Call Basket Price: %8.4f\n'  , QuasiMonteCarloCallPrice)
```

```
 Quasi-Monte Carlo American Call Basket Price:  21.8202
```

```
fprintf(' Quasi-Monte Carlo American Put  Basket Price: %8.4f\n\n', QuasiMonteCarloPutPrice)
```

```
 Quasi-Monte Carlo American Put  Basket Price:  19.7968
```

```
fprintf(' Gaussian Copula American Call Basket Price: %8.4f\n'  , GaussianCopulaCallPrice)
```

```
 Gaussian Copula American Call Basket Price:  24.5711
```

```
fprintf(' Gaussian Copula American Put  Basket Price: %8.4f\n\n', GaussianCopulaPutPrice)
```

```
 Gaussian Copula American Put  Basket Price:  17.4229
```

```
fprintf('        t Copula American Call Basket Price: %8.4f\n'  , tCopulaCallPrice)
```

```
        t Copula American Call Basket Price:  22.6220
```

```
fprintf('        t Copula American Put  Basket Price: %8.4f\n'  , tCopulaPutPrice)
```

```
        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:  20.0945
Brownian Motion American Put  Basket Price:  16.4808
```

```
Quasi-Monte Carlo American Call Basket Price:  20.1935
Quasi-Monte Carlo American Put  Basket Price:  16.4731

Gaussian Copula American Call Basket Price:  20.9183
Gaussian Copula American Put  Basket Price:  16.4416

       t Copula American Call Basket Price:  21.0133
       t Copula American Put  Basket Price:  16.7050
```

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)

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', dt, ...
                          'Processes', f.LongstaffSchwartz, ...
                          'MontecarloMethod','quasi');

QuasiMonteCarloCallPrice = f.CallPrice(strike, r/252);
QuasiMonteCarloPutPrice  = 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

sde | bm | gbm | merton | bates | drift | diffusion | sdeddo | sdeld | cev | cir | heston | hwv | sdemrd | ts2func | simulate | simByEuler | simBySolution | simBySolution | interpolate | simByQuadExp

Related Examples

- “Simulating Equity Prices” on page 14-28
- “Simulating Interest Rates” on page 14-48
- “Improving Performance of Monte Carlo Simulation with Parallel Computing” on page 14-87
- “Base SDE Models” on page 14-14
- “Drift and Diffusion Models” on page 14-16
- “Linear Drift Models” on page 14-19
- “Parametric Models” on page 14-21

More About

- “SDEs” on page 14-2
- “SDE Models” on page 14-7
- “SDE Class Hierarchy” on page 14-5
- “Quasi-Monte Carlo Simulation” on page 14-62

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 that you want to include a time-varying short rate and 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 are obtained by linear interpolation. For example, such a simulation can 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. Also, 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 `gbm`. 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 the `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

`sde` | `bm` | `gbm` | `merton` | `bates` | `drift` | `diffusion` | `sdeddo` | `sdeld` | `cev` | `cir` | `heston` | `hwv` | `sdemrd` | `ts2func` | `simulate` | `simByEuler` | `simBySolution` | `simBySolution` | `interpolate` | `simByQuadExp`

Related Examples

- “Simulating Equity Prices” on page 14-28
- “Simulating Interest Rates” on page 14-48
- “Price American Basket Options Using Standard Monte Carlo and Quasi-Monte Carlo Simulation” on page 14-70
- “Base SDE Models” on page 14-14
- “Drift and Diffusion Models” on page 14-16
- “Linear Drift Models” on page 14-19
- “Parametric Models” on page 14-21

More About

- “SDEs” on page 14-2
- “SDE Models” on page 14-7
- “SDE Class Hierarchy” on page 14-5
- “Performance Considerations” on page 14-64

15

Functions

brinsonAttribution

Create brinsonAttribution object to analyze performance attribution

Description

Create a brinsonAttribution object for performance attribution using the Brinson model.

Use this workflow to develop and analyze a Brinson model for performance attribution:

1. Prepare data for the AssetTable input.
2. Use brinsonAttribution to create a brinsonAttribution object.
3. Use the following functions with the brinsonAttribution object:

   - categoryAttribution
   - categoryReturns
   - categoryWeights
   - totalAttribution
   - summary
   - categoryReturnsChart
   - categoryWeightsChart
   - attributionsChart

For more detailed information on this workflow, see “Analyze Performance Attribution Using Brinson Model” on page 4-320.

Creation

Syntax

```
brinsonAttrubtionObj = brinsonAttribution(AssetTable)
```

Description

brinsonAttrubtionObj = brinsonAttribution(AssetTable) creates a brinsonAttribution object and sets the properties on page 15-3. Use the brinsonAttribution object with the categoryAttribution, categoryReturns, categoryWeights, totalAttribution, and summary functions.

Input Arguments

AssetTable — Information about individual asset returns in the portfolio and benchmark
table

Information about individual asset returns in the portfolio and benchmark, specified as a table with the number of rows equal to NumAssets-times-NumPeriods rows. The column variables are in the following order from left to right:

- Period — Column vector of positive whole numbers containing the time Period numbers. For each one of the NumAssets asset names, the Period numbers range from 1 to NumPeriods with increments of 1, so that there are a total of NumAssets-times-NumPeriods rows. Row 1 is for the first period, row 2 is for the second period, and is repeated for each asset. If the Period column is missing from AssetTable, all Period numbers are internally set to 1 and all asset returns in AssetTable are assumed to be for the same single period.
- Name — String column vector containing the individual asset names for the associated returns. There is a total of NumAssets unique names, and the names are repeated for each Period.
- Return — Numeric column vector containing the asset returns in decimals.
- Category — Categorical vector containing the asset categories (sectors) for the associated asset returns. There is a total of NumCategories unique categories.
- Portfolio Weight — Numeric vector containing the asset portfolio weights in decimals. The weights are internally normalized so that they sum to 1 for each Period.

  Note The values for Portfolio Weight must sum to 1 for the Brinson model. If the weights do not sum to 1, the Portfolio Weight values are internally normalized so that they sum to 1 and brinsonAttribution displays a warning. If there is a cash position, you must account for it as an asset with its own weight so that the weights sum to 1.
- Benchmark Weight — Numeric vector containing the asset benchmark weights in decimals. The weights are internally normalized so that they sum to 1 for each Period.

  Note The values for the Benchmark Weight must sum to 1 for the Brinson model. If the weights do not sum to 1, the Benchmark Weight values are internally normalized so that they sum to 1 and brinsonAttribution displays a warning. If there is a cash position, you must account for it as an asset with its own weight so that the weights sum to 1.

Note AssetTable must be a table. Instead of specific dates and times, the Period column of AssetTable must have period numbers that start with 1 and have increments of one. The multiperiod Brinson model assumes that all returns are for time periods of equal intervals (for example, monthly, quarterly, and so on).

Data Types: table

Properties

NumAssets — Total number of assets
numeric

Total number of assets, specified by AssetTable.

Data Types: double

NumPortfolioAssets — Number of assets in portfolio
numeric

Number of assets in the portfolio, specified by AssetTable.

Data Types: double

NumBenchmarkAssets — Number of assets in the benchmark
numeric

Number of assets in the benchmark, specified by AssetTable.

Data Types: double

NumPeriods — Number of time periods
numeric

Number of time periods, specified by AssetTable.

Data Types: double

NumCategories — Number of asset categories
numeric

Number of asset categories, specified by the AssetTable.

Data Types: double

AssetName — Individual asset names
vector

Individual asset names, returned as a NumAssets-by-1 vector.

Data Types: string

AssetReturn — Asset returns
matrix

Asset returns, returned as a NumAssets-by-NumPeriods matrix in decimals.

Data Types: double

AssetCategory — Asset categories
matrix

Asset categories (sectors), returned as a NumAssets-by-NumPeriods matrix.

Data Types: string

PortfolioAssetWeight — Asset portfolio weights
numeric

Asset portfolio weights, returned as a NumAssets-by-NumPeriods matrix in decimals.

Data Types: double

BenchmarkAssetWeight — Asset benchmark weights
numeric

Asset benchmark weights, returned as a NumAssets-by-NumPeriods matrix in decimals.

Data Types: double

PortflioCategoryReturn — Portfolio category returns
numeric

Portfolio category returns, returned as a `NumCategories`-by-`NumPeriods` matrix in decimals.

Data Types: `double`

`BenchmarkCategoryReturn` — Benchmark category returns
numeric

Benchmark category returns, returned as a `NumCategories`-by-`NumPeriods` matrix in decimals.

Data Types: `double`

`PortfolioCategoryWeight` — Portfolio category weights
numeric

Portfolio category weights, returned as a `NumCategories`-by-`NumPeriods` matrix in decimals.

Data Types: `double`

`BenchmarkCategoryWeight` — Benchmark category weights
numeric

Benchmark category weights, returned as a `NumCategories`-by-`NumPeriods` matrix in decimals.

Data Types: `double`

`PortfolioReturn` — Total portfolio return
numeric

Total portfolio return, returned as a scalar decimal.

Data Types: `double`

`BenchmarkReturn` — Total benchmark return
numeric

Total benchmark return, returned as a scalar decimal.

Data Types: `double`

`ActiveReturn` — Total active return
numeric

Total active return, returned as a scalar decimal.

Data Types: `double`

Object Functions

categoryAttribution	Compute performance attribution for portfolio of each category
categoryReturns	Compute aggregate and periodic category returns
categoryWeights	Compute average and periodic category weights
totalAttribution	Compute total performance attribution by Brinson model
summary	Summarize performance attribution by Brinson model
categoryReturnsChart	Create horizontal bar chart of category returns
categoryWeightsChart	Create a horizontal bar chart for category weights
attributionsChart	Create horizontal bar chart of performance attribution

Examples

Create brinsonAttribution Object

This example shows how to create a `brinsonAttribution` object that you can then use with the `categoryAttribution`, `categoryReturns`, `categoryWeights`, `totalAttribution`, and `summary` functions. Also, you can generate plots for the results, using `categoryReturnsChart`, `categoryWeightsChart`, and `attributionsChart`.

Prepare Data

Create a table for the monthly prices for four assets.

```
 GM =[17.82;22.68;19.37;20.28];
 HD = [39.79;39.12;40.67;40.96];
 KO = [38.98;39.44;40.00;40.20];
 PG = [56.38;57.08;57.76;55.54];
 MonthlyPrices  = table(GM,HD,KO,PG);
```

Use `tick2ret` to define the monthly returns.

```
MonthlyReturns = tick2ret(MonthlyPrices.Variables)';
  [NumAssets,NumPeriods] = size(MonthlyReturns);
```

Define the periods.

```
Period = ones(NumAssets*NumPeriods,1);
  for k = 1:NumPeriods
      Period(k*NumAssets+1:end,1) = Period(k*NumAssets,1) + 1;
  end
```

Define the categories (sectors) for the four assets.

```
Name = repmat(string(MonthlyPrices.Properties.VariableNames(:)),NumPeriods,1);
  Categories = repmat(categorical([ ...
      "Consumer Discretionary"; ...
      "Consumer Discretionary"; ...
      "Consumer Staples"; ...
      "Consumer Staples"]),NumPeriods,1);
```

Define benchmark and portfolio weights.

```
BenchmarkWeight = repmat(1./NumAssets.*ones(NumAssets, 1),NumPeriods,1);
  PortfolioWeight = repmat([1;0;1;1]./3,NumPeriods,1);
```

Create AssetTable Input

Create `AssetTable` as the input for the `brinsonAttribution` object.

```
  AssetTable = table(Period, Name, ...
      MonthlyReturns(:), Categories, PortfolioWeight, BenchmarkWeight, ...
      VariableNames=["Period","Name","Return","Category","PortfolioWeight","BenchmarkWeight"])
```

```
AssetTable=12×6 table
    Period    Name    Return              Category              PortfolioWeight    BenchmarkWeight
    ______    ____    _________    ______________________    _______________    _______________
```

```
      1       "GM"      0.27273    Consumer Discretionary        0.33333            0.25
      1       "HD"    -0.016838    Consumer Discretionary              0            0.25
      1       "KO"     0.011801    Consumer Staples              0.33333            0.25
      1       "PG"     0.012416    Consumer Staples              0.33333            0.25
      2       "GM"     -0.14594    Consumer Discretionary        0.33333            0.25
      2       "HD"     0.039622    Consumer Discretionary              0            0.25
      2       "KO"     0.014199    Consumer Staples              0.33333            0.25
      2       "PG"     0.011913    Consumer Staples              0.33333            0.25
      3       "GM"      0.04698    Consumer Discretionary        0.33333            0.25
      3       "HD"    0.0071306    Consumer Discretionary              0            0.25
      3       "KO"        0.005    Consumer Staples              0.33333            0.25
      3       "PG"    -0.038435    Consumer Staples              0.33333            0.25
```

Create brinsonAttribution Object

Use `brinsonAttribution` to create the `brinsonAttribution` object.

```
  BrinsonPAobj = brinsonAttribution(AssetTable)
```

```
BrinsonPAobj = 
  brinsonAttribution with properties:

                  NumAssets: 4
         NumPortfolioAssets: 3
         NumBenchmarkAssets: 4
                 NumPeriods: 3
              NumCategories: 2
                  AssetName: [4x1 string]
                AssetReturn: [4x3 double]
              AssetCategory: [4x3 categorical]
       PortfolioAssetWeight: [4x3 double]
       BenchmarkAssetWeight: [4x3 double]
    PortfolioCategoryReturn: [2x3 double]
    BenchmarkCategoryReturn: [2x3 double]
    PortfolioCategoryWeight: [2x3 double]
    BenchmarkCategoryWeight: [2x3 double]
            PortfolioReturn: 0.0598
            BenchmarkReturn: 0.0540
               ActiveReturn: 0.0059
```

You use `brinsonAttribution` object with the `categoryAttribution`, `categoryReturns`, `categoryWeights`, `totalAttribution`, and `summary` functions for the anlaysis. Also, you can generate plots for the results, using `categoryReturnsChart`, `categoryWeightsChart`, and `attributionsChart`.

More About

Brinson Model

Performance attribution supports methods for single periods over relatively short time spans (monthly) and multiple periods compounded over longer time spans (quarterly or a yearly).

The Brinson single-period sector-based (category-based) performance attribution is represented as:

$$
\begin{aligned}
r_{P,t} &= \sum_{j=1}^{N} w_{Pj,t} r_{Pj,t} \\
r_{B,t} &= \sum_{j=1}^{N} w_{Bj,t} r_{Bj,t} \\
r_{\upsilon,t} = r_{P,t} - r_{B,t} &= \sum_{j=1}^{N} w_{Pj,t} r_{Pj,t} - \sum_{j=1}^{N} w_{Bj,t} r_{Bj,t} \\
&= \sum_{j=1}^{N} (w_{Pj,t} - w_{Bj,t})(r_{Bj,t} - r_{B,t}) + \sum_{j=1}^{N} (w_{Pj,t} - w_{Bj,t})(r_{Pj,t} - r_{Bj,t}) + \sum_{j=1}^{N} w_{Bj,t}(r_{Pj,t} - r_{Bj,t}) \\
&= \sum_{j=1}^{N} (S_{j,t} + U_{j,t} + I_{j,t})
\end{aligned}
$$

where

$r_{P,t}$ — Portfolio return for period t

$r_{b,t}$ — Benchmark return for period t

$r_{\upsilon,t}$ — Value-added return for period t

N — Number of sectors

$w_{Pj,t}$ — Portfolio weight for sector j and period t

$r_{Pj,t}$ — Portfolio return for sector j and period t

$w_{Bj,t}$ — Benchmark weight for sector j and period t

$r_{Bj,t}$ — Benchmark return for sector j and period t

$S_{j,t}$ — Sector allocation effect for sector j and period t

$U_{j,t}$ — Allocation and sector interaction effect for sector j and period t

$I_{j,t}$ — Issue sector effect for sector j and period t

Multiperiod performance attribution, uses two main styles of performance attribution: arithmetic and geometric. In the arithmetic performance attribution, which is the method that the Brinson model uses, the relative performance between the portfolio and benchmark is measured by subtracting the benchmark return from the portfolio return. In the geometric performance attribution, the relative performance between the portfolio and benchmark is measured by a ratio based on the portfolio and benchmark returns.

The optimized linking algorithm by Menchero [3] extends the single-period arithmetic performance attribution to multiple periods by introducing the optimized linking coefficient β_t for each period before adding the single-period attributions. This step overcomes the fact that each single-period attribution does not simply add over multiple periods, because the optimized linking coefficient β_t reconciles the difference that arises between simple arithmetic addition and geometric compounding. As a result, the single-period Brinson model can be extended to multiple periods by using the following optimized linking algorithm:

$$r_v = r_p - r_B = \sum_{t=1}^{T} \beta_t (r_{P,t} - r_{B,t}) = \sum_{j=1}^{N} \left(\sum_{t=1}^{T} \beta_t S_{j,t} + \sum_{t=1}^{T} \beta_t U_{j,t} + \sum_{t=1}^{T} \beta_t I_{j,t} \right) = \sum_{j=1}^{N} (S_j + U_j + I_j)$$

$$\beta_t = A + C(r_{p,t} - r_{B,t})$$

$$A = \frac{\frac{(r_p - r_B)}{T}}{(1 + r_p)^{\frac{1}{T}} - (1 + r_B)^{\frac{1}{T}}}$$

$$C = \frac{r_p - r_B - A\sum_{t=1}^{T}(r_{P,t} - r_{B,t})}{\sum_{t=1}^{T}(r_{P,t} - r_{B,t})^2}$$

where

r_p — Portfolio return over multiple periods

r_B — Benchmark return over multiple periods

r_v — Value-added return over multiple periods

$r_{P,t}$ — Portfolio return for period t

$r_{B,t}$ — Benchmark return for period t

β_t — Optimized linking coefficient for period t

T — Number of periods

N — Number of sectors

$S_{j,t}$ — Sector allocation effect for sector j and period t

$U_{j,t}$ — Allocation and selection interaction effect for sector j and period t

$I_{j,t}$ — Issue selection effect for sector j and period t

$S_{j,t}$ — Sector allocation effect for sector j over multiple periods

$U_{j,t}$ — Allocation and sector interaction effect for sector j over multiple periods

$I_{j,t}$ — Issue sector effect for sector j over multiple periods

In Menchero's optimized linking algorithm shown above, the optimized linking coefficient β_t is computed for each period using r_P, r_B, $r_{P,t}$, $r_{B,t}$, and T. This algorithm allows extending the Brinson model to multiple periods while achieving the resulting desirable properties that are similar to the geometric multiperiod performance attribution.

Version History

Introduced in R2022b

R2023a: Charting functions added for `brinsonAttribution` object

Using a `brinsonAttribution` object, you can use the following charting functions: `categoryReturnsChart`, `categoryWeightsChart`, and `attributesChart`.

References

[1] Brinson, G. P. and Fachler, N. “Measuring Non-US Equity Portfolio Performance.” *Journal of Portfolio Management.* Spring 1985: 73–76.

[2] Brinson, G. P., Hood, L. R., and Beebower, G. L. “Determinants of Portfolio Performance.” *Financial Analysts Journal.* Vol. 42, No. 4, 1986: 39–44.

[3] Menchero, J. “Multiperiod Arithmetic Attribution.” *Financial Analysts Journal.* Vol. 60, No. 4, 2004: 76–91.

[4] Tuttle, D. L., Pinto, J. E., and McLeavey, D. W. *Managing Investment Portfolios: A Dynamic Process.* Third Edition. CFA Institute, 2007.

See Also

`table`

Topics

“Analyze Performance Attribution Using Brinson Model” on page 4-320
“Backtest with Brinson Attribution to Evaluate Portfolio Performance” on page 4-311

External Websites

Backtesting Strategy Framework in Financial Toolbox (2 min 17 sec)

categoryAttribution

Compute performance attribution for portfolio of each category

Syntax

```
[AggregateCategoryAttribution,PeriodicCategoryAttribution] =
categoryAttribution(brinsonAttributionObj)
```

Description

`[AggregateCategoryAttribution,PeriodicCategoryAttribution] = categoryAttribution(brinsonAttributionObj)` computes the performance attribution for the portfolio of each category using the Brinson model.

Examples

Compute Performance Attribution Using `brinsonAttribution` Object

This example shows how to create a `brinsonAttribution` object and then use `categoryAttribution` to compute the performance attribution of the portfolio for each category (sector).

Prepare Data

Create a table for the monthly prices for four assets.

```
GM =[17.82;22.68;19.37;20.28];
 HD = [39.79;39.12;40.67;40.96];
 KO = [38.98;39.44;40.00;40.20];
 PG = [56.38;57.08;57.76;55.54];
 MonthlyPrices  = table(GM,HD,KO,PG);
```

Use `tick2ret` to define the monthly returns.

```
 MonthlyReturns = tick2ret(MonthlyPrices.Variables)';
 [NumAssets,NumPeriods] = size(MonthlyReturns);
```

Define the periods.

```
 Period = ones(NumAssets*NumPeriods,1);
 for k = 1:NumPeriods
     Period(k*NumAssets+1:end,1) = Period(k*NumAssets,1) + 1;
 end
```

Define the categories for the four assets.

```
 Name = repmat(string(MonthlyPrices.Properties.VariableNames(:)),NumPeriods,1);
 Categories = repmat(categorical([ ...
     "Consumer Discretionary"; ...
     "Consumer Discretionary"; ...
```

```
    "Consumer Staples"; ...
    "Consumer Staples"]),NumPeriods,1);
```

Define benchmark and portfolio weights.

```
BenchmarkWeight = repmat(1./NumAssets.*ones(NumAssets, 1),NumPeriods,1);
PortfolioWeight = repmat([1;0;1;1]./3,NumPeriods,1);
```

Create AssetTable Input

Create `AssetTable` as the input for the `brinsonAttribution` object.

```
AssetTable = table(Period, Name, ...
     MonthlyReturns(:), Categories, PortfolioWeight, BenchmarkWeight, ...
     VariableNames=["Period","Name","Return","Category","PortfolioWeight","BenchmarkWeight"])
```

```
AssetTable=12×6 table
    Period    Name     Return              Category              PortfolioWeight    BenchmarkWeight
    ______    ____    _________    ______________________    _______________    _______________

      1       "GM"      0.27273    Consumer Discretionary        0.33333             0.25
      1       "HD"    -0.016838    Consumer Discretionary              0             0.25
      1       "KO"     0.011801    Consumer Staples              0.33333             0.25
      1       "PG"     0.012416    Consumer Staples              0.33333             0.25
      2       "GM"     -0.14594    Consumer Discretionary        0.33333             0.25
      2       "HD"     0.039622    Consumer Discretionary              0             0.25
      2       "KO"     0.014199    Consumer Staples              0.33333             0.25
      2       "PG"     0.011913    Consumer Staples              0.33333             0.25
      3       "GM"      0.04698    Consumer Discretionary        0.33333             0.25
      3       "HD"    0.0071306    Consumer Discretionary              0             0.25
      3       "KO"        0.005    Consumer Staples              0.33333             0.25
      3       "PG"    -0.038435    Consumer Staples              0.33333             0.25
```

Create brinsonAttribution Object

Use `brinsonAttribution` to create the `brinsonAttribution` object.

```
BrinsonPAobj = brinsonAttribution(AssetTable)
```

```
BrinsonPAobj = 
  brinsonAttribution with properties:

                  NumAssets: 4
         NumPortfolioAssets: 3
         NumBenchmarkAssets: 4
                 NumPeriods: 3
              NumCategories: 2
                  AssetName: [4x1 string]
                AssetReturn: [4x3 double]
              AssetCategory: [4x3 categorical]
       PortfolioAssetWeight: [4x3 double]
       BenchmarkAssetWeight: [4x3 double]
    PortfolioCategoryReturn: [2x3 double]
    BenchmarkCategoryReturn: [2x3 double]
    PortfolioCategoryWeight: [2x3 double]
    BenchmarkCategoryWeight: [2x3 double]
            PortfolioReturn: 0.0598
            BenchmarkReturn: 0.0540
```

```
           ActiveReturn: 0.0059
```

Compute Performance Attribution

Use the `brinsonAttribution` object with `categoryAttribution` to compute the performance attribution of the portfolio for each category.

```
[AggregateCategoryAttribution,PeriodicCategoryAttribution] = categoryAttribution(BrinsonPAobj)
```

```
AggregateCategoryAttribution=2×5 table
           Category           Allocation     Selection     Interaction    ActiveReturn
    ______________________    __________    ___________    ___________    ____________

    Consumer Discretionary    -0.0071764       0.030316      -0.010105       0.013034 
    Consumer Staples          -0.0071764    -9.4005e-19    -3.1335e-19     -0.0071764 


PeriodicCategoryAttribution=6×7 table
    Period    LinkingCoefficient           Category           Allocation     Selection     Intera
    ______    __________________    ______________________    __________    ___________    ______

      1            0.97032          Consumer Discretionary     -0.009653       0.072391       -0
      1            0.97032          Consumer Staples           -0.009653              0          
      2             1.0838          Consumer Discretionary     0.0055181      -0.046391       0.0
      2             1.0838          Consumer Staples           0.0055181    -8.6736e-19    -2.891
      3             1.0391          Consumer Discretionary    -0.0036477      0.0099623     -0.00
      3             1.0391          Consumer Staples          -0.0036477              0          
```

Input Arguments

`brinsonAttributionObj` — Brinson attribution model
object

Brinson attribution model, specified as a `brinsonAttribution` object.

Data Types: `object`

Output Arguments

`AggregateCategoryAttribution` — Category attribution aggregated over all periods
table

Category attribution aggregated over all periods, returned as a table with the following columns:

- `Category` — Asset category
- `Allocation` — Category allocation effect for the category
- `Selection` — Effect of selecting individual assets within the category
- `Interaction` — Allocation-selection interaction effect
- `ActiveReturn` — Active return compared with benchmark

`PeriodicCategoryAttribution` — Category attribution for each period
table

Category attribution for each period, returned as a table with the following columns:

- `Period` — Time period numbers (1 for the first period, 2 for the second period, and so on)
- `LinkingCoefficient` — Linking coefficients used to aggregate the attribution results over all time periods
- `Category` — Asset category
- `Allocation` — Category allocation effect for the category
- `Selection` — Effect of selecting individual assets within the category
- `Interaction` — Allocation-selection interaction effect
- `ActiveReturn` — Active return compared with benchmark

Version History

Introduced in R2022b

References

[1] Brinson, G. P. and Fachler, N. “Measuring Non-US Equity Portfolio Performance.” *Journal of Portfolio Management.* Spring 1985: 73–76.

[2] Brinson, G. P., Hood, L. R., and Beebower, G. L. “Determinants of Portfolio Performance.” *Financial Analysts Journal.* Vol. 42, No. 4, 1986: 39–44.

[3] Menchero, J. “Multiperiod Arithmetic Attribution.” *Financial Analysts Journal.* Vol. 60, No. 4, 2004: 76–91.

[4] Tuttle, D. L., Pinto, J. E., and McLeavey, D. W. *Managing Investment Portfolios: A Dynamic Process.* Third Edition. CFA Institute, 2007.

See Also

`totalAttribution` | `categoryWeights` | `categoryReturns` | `summary` | `categoryReturnsChart` | `categoryWeightsChart` | `attributionsChart`

Topics

“Analyze Performance Attribution Using Brinson Model” on page 4-320
“Backtest with Brinson Attribution to Evaluate Portfolio Performance” on page 4-311

categoryReturns

Compute aggregate and periodic category returns

Syntax

`[AggregateCategoryReturns,PeriodicCategoryReturns] = categoryReturns( brinsonAttributionObj)`

Description

`[AggregateCategoryReturns,PeriodicCategoryReturns] = categoryReturns( brinsonAttributionObj)` computes the aggregate and periodic category (sector) returns for the portfolio and the benchmark.

Examples

Compute Category Returns Using `brinsonAttribution` Object

This example shows how to create a `brinsonAttribution` object and then use `categoryReturns` to compute the aggregate and periodic category returns for the portfolio and the benchmark.

Prepare Data

Create a table for the monthly prices for four assets.

```
 GM =[17.82;22.68;19.37;20.28];
 HD = [39.79;39.12;40.67;40.96];
 KO = [38.98;39.44;40.00;40.20];
 PG = [56.38;57.08;57.76;55.54];
 MonthlyPrices  = table(GM,HD,KO,PG);
```

Use `tick2ret` to define the monthly returns.

```
MonthlyReturns = tick2ret(MonthlyPrices.Variables)';
  [NumAssets,NumPeriods] = size(MonthlyReturns);
```

Define the periods.

```
Period = ones(NumAssets*NumPeriods,1);
  for k = 1:NumPeriods
      Period(k*NumAssets+1:end,1) = Period(k*NumAssets,1) + 1;
  end
```

Define the categories for the four assets.

```
Name = repmat(string(MonthlyPrices.Properties.VariableNames(:)),NumPeriods,1);
  Categories = repmat(categorical([ ...
      "Consumer Discretionary"; ...
      "Consumer Discretionary"; ...
      "Consumer Staples"; ...
      "Consumer Staples"]),NumPeriods,1);
```

Define benchmark and portfolio weights.

```
BenchmarkWeight = repmat(1./NumAssets.*ones(NumAssets, 1),NumPeriods,1);
  PortfolioWeight = repmat([1;0;1;1]./3,NumPeriods,1);
```

Create AssetTable Input

Create `AssetTable` as the input for the `brinsonAttribution` object.

```
  AssetTable = table(Period, Name, ...
      MonthlyReturns(:), Categories, PortfolioWeight, BenchmarkWeight, ...
      VariableNames=["Period","Name","Return","Category","PortfolioWeight","BenchmarkWeight"])
```

```
AssetTable=12×6 table
    Period    Name     Return             Category            PortfolioWeight    BenchmarkWeight
    ______    ____    _________    ______________________    _______________    _______________

      1       "GM"      0.27273    Consumer Discretionary        0.33333             0.25
      1       "HD"    -0.016838    Consumer Discretionary              0             0.25
      1       "KO"     0.011801    Consumer Staples              0.33333             0.25
      1       "PG"     0.012416    Consumer Staples              0.33333             0.25
      2       "GM"     -0.14594    Consumer Discretionary        0.33333             0.25
      2       "HD"     0.039622    Consumer Discretionary              0             0.25
      2       "KO"     0.014199    Consumer Staples              0.33333             0.25
      2       "PG"     0.011913    Consumer Staples              0.33333             0.25
      3       "GM"      0.04698    Consumer Discretionary        0.33333             0.25
      3       "HD"    0.0071306    Consumer Discretionary              0             0.25
      3       "KO"        0.005    Consumer Staples              0.33333             0.25
      3       "PG"    -0.038435    Consumer Staples              0.33333             0.25
```

Create brinsonAttribution Object

Use `brinsonAttribution` with the `brinsonAttribution` object to compute aggregate and periodic category returns.

```
  BrinsonPAobj = brinsonAttribution(AssetTable)
```

```
BrinsonPAobj = 
  brinsonAttribution with properties:

                  NumAssets: 4
         NumPortfolioAssets: 3
         NumBenchmarkAssets: 4
                 NumPeriods: 3
              NumCategories: 2
                  AssetName: [4x1 string]
                AssetReturn: [4x3 double]
              AssetCategory: [4x3 categorical]
       PortfolioAssetWeight: [4x3 double]
       BenchmarkAssetWeight: [4x3 double]
    PortfolioCategoryReturn: [2x3 double]
    BenchmarkCategoryReturn: [2x3 double]
    PortfolioCategoryWeight: [2x3 double]
    BenchmarkCategoryWeight: [2x3 double]
            PortfolioReturn: 0.0598
            BenchmarkReturn: 0.0540
               ActiveReturn: 0.0059
```

Compute Category Returns

Use `categoryReturns` with the `brinsonAttribution` object to compute aggregate and periodic category (sector) returns.

```
[AggregateCategoryReturns, PeriodicCategoryReturns] = categoryReturns(BrinsonPAobj)
```

```
AggregateCategoryReturns=2×3 table
           Category            AggregatePortfolioReturn    AggregateBenchmarkReturn
    ______________________    ________________________    ________________________

    Consumer Discretionary             0.13805                     0.096876        
    Consumer Staples                 0.0081816                    0.0081816        


PeriodicCategoryReturns=6×4 table
    Period           Category            PortfolioReturn    BenchmarkReturn
    ______    ______________________    _______________    _______________

      1       Consumer Discretionary        0.27273            0.12794     
      1       Consumer Staples             0.012108           0.012108     
      2       Consumer Discretionary       -0.14594          -0.053161     
      2       Consumer Staples             0.013056           0.013056     
      3       Consumer Discretionary        0.04698           0.027055     
      3       Consumer Staples            -0.016717          -0.016717     
```

Input Arguments

`brinsonAttributionObj` — Brinson attribution model

object

Brinson attribution model, specified as a `brinsonAttribution` object.

Data Types: `object`

Output Arguments

`AggregateCategoryReturns` — Category returns aggregated over all periods

table

Category returns aggregated over all periods, returned as a table with the following columns:

- `Category` — Asset category
- `AggregatePortfolioReturn` — Aggregate portfolio returns
- `AggregateBenchmarkReturn` — Aggregate benchmark returns

`PeriodicCategoryReturns` — Category returns for each period

table

Category returns for each period, returned as a table with the following columns:

- `Period` — Time period numbers (1 for the first period, 2 for the second period, and so on)
- `Category` — Asset category

- `PortfolioReturn` — Portfolio returns
- `BenchmarkReturn` — Benchmark returns

Version History

Introduced in R2020b

References

[1] Brinson, G. P. and Fachler, N. “Measuring Non-US Equity Portfolio Performance.” *Journal of Portfolio Management.* Spring 1985: 73–76.

[2] Brinson, G. P., Hood, L. R., and Beebower, G. L. “Determinants of Portfolio Performance.” *Financial Analysts Journal.* Vol. 42, No. 4, 1986: 39–44.

[3] Menchero, J. “Multiperiod Arithmetic Attribution.” *Financial Analysts Journal.* Vol. 60, No. 4, 2004: 76–91.

[4] Tuttle, D. L., Pinto, J. E., and McLeavey, D. W. *Managing Investment Portfolios: A Dynamic Process.* Third Edition. CFA Institute, 2007.

See Also

`totalAttribution` | `categoryWeights` | `categoryAttribution` | `summary` | `categoryReturnsChart` | `categoryWeightsChart` | `attributesChart`

Topics

“Analyze Performance Attribution Using Brinson Model” on page 4-320
“Backtest with Brinson Attribution to Evaluate Portfolio Performance” on page 4-311

categoryWeights

Compute average and periodic category weights

Syntax

```
[AverageCategoryWeights,PeriodicCategoryWeights] = categoryWeights(
brinsonAttributionObj)
```

Description

`[AverageCategoryWeights,PeriodicCategoryWeights] = categoryWeights( brinsonAttributionObj)` computes the average and periodic category weights for the portfolio and the benchmark as well as the corresponding active weights.

Examples

Compute Category Weights Using `brinsonAttribution` Object

This example shows how to create a `brinsonAttribution` object and then use `categoryWeights` to compute average and periodic category (sector) weights.

Prepare Data

Create a table for the monthly prices for four assets.

```
 GM =[17.82;22.68;19.37;20.28];
 HD = [39.79;39.12;40.67;40.96];
 KO = [38.98;39.44;40.00;40.20];
 PG = [56.38;57.08;57.76;55.54];
 MonthlyPrices  = table(GM,HD,KO,PG);
```

Use `tick2ret` to define the monthly returns.

```
MonthlyReturns = tick2ret(MonthlyPrices.Variables)';
  [NumAssets,NumPeriods] = size(MonthlyReturns);
```

Define the periods.

```
Period = ones(NumAssets*NumPeriods,1);
  for k = 1:NumPeriods
      Period(k*NumAssets+1:end,1) = Period(k*NumAssets,1) + 1;
  end
```

Define the categories for the four assets.

```
Name = repmat(string(MonthlyPrices.Properties.VariableNames(:)),NumPeriods,1);
  Categories = repmat(categorical([ ...
      "Consumer Discretionary"; ...
      "Consumer Discretionary"; ...
      "Consumer Staples"; ...
      "Consumer Staples"]),NumPeriods,1);
```

Define benchmark and portfolio weights.

```
BenchmarkWeight = repmat(1./NumAssets.*ones(NumAssets, 1),NumPeriods,1);
  PortfolioWeight = repmat([1;0;1;1]./3,NumPeriods,1);
```

Create AssetTable Input

Create `AssetTable` as the input for the `brinsonAttribution` object.

```
  AssetTable = table(Period, Name, ...
      MonthlyReturns(:), Categories, PortfolioWeight, BenchmarkWeight, ...
      VariableNames=["Period","Name","Return","Category","PortfolioWeight","BenchmarkWeight"])
```

```
AssetTable=12×6 table
    Period    Name      Return              Category            PortfolioWeight    BenchmarkWeight
    ______    ____    _________    ______________________    _______________    _______________

      1       "GM"      0.27273    Consumer Discretionary        0.33333             0.25      
      1       "HD"    -0.016838    Consumer Discretionary              0             0.25      
      1       "KO"     0.011801    Consumer Staples              0.33333             0.25      
      1       "PG"     0.012416    Consumer Staples              0.33333             0.25      
      2       "GM"     -0.14594    Consumer Discretionary        0.33333             0.25      
      2       "HD"     0.039622    Consumer Discretionary              0             0.25      
      2       "KO"     0.014199    Consumer Staples              0.33333             0.25      
      2       "PG"     0.011913    Consumer Staples              0.33333             0.25      
      3       "GM"      0.04698    Consumer Discretionary        0.33333             0.25      
      3       "HD"    0.0071306    Consumer Discretionary              0             0.25      
      3       "KO"        0.005    Consumer Staples              0.33333             0.25      
      3       "PG"    -0.038435    Consumer Staples              0.33333             0.25      
```

Create brinsonAttribution Object

Use `brinsonAttribution` to create the `brinsonAttribution` object.

```
  BrinsonPAobj = brinsonAttribution(AssetTable)
```

```
BrinsonPAobj = 
  brinsonAttribution with properties:

                    NumAssets: 4
           NumPortfolioAssets: 3
           NumBenchmarkAssets: 4
                   NumPeriods: 3
                NumCategories: 2
                    AssetName: [4x1 string]
                  AssetReturn: [4x3 double]
                AssetCategory: [4x3 categorical]
         PortfolioAssetWeight: [4x3 double]
         BenchmarkAssetWeight: [4x3 double]
      PortfolioCategoryReturn: [2x3 double]
      BenchmarkCategoryReturn: [2x3 double]
      PortfolioCategoryWeight: [2x3 double]
      BenchmarkCategoryWeight: [2x3 double]
              PortfolioReturn: 0.0598
              BenchmarkReturn: 0.0540
                 ActiveReturn: 0.0059
```

Compute Category Weights

Use `categoryWeights` to compute the average and periodic category weights for the portfolio and the benchmark, as well as, the corresponding active weights.

```
[AverageCategoryWeights,PeriodicCategoryWeights] = categoryWeights(BrinsonPAobj)
```

```
AverageCategoryWeights=2×4 table
           Category            AveragePortfolioWeight    AverageBenchmarkWeight    AverageActiveWe
    ______________________    ______________________    ______________________    _______________

    Consumer Discretionary           0.33333                      0.5                   -0.16667
    Consumer Staples                 0.66667                      0.5                    0.16667

PeriodicCategoryWeights=6×5 table
    Period           Category            PortfolioWeight    BenchmarkWeight    ActiveWeight
    ______    ______________________    _______________    _______________    ____________

      1       Consumer Discretionary        0.33333              0.5            -0.16667
      1       Consumer Staples              0.66667              0.5             0.16667
      2       Consumer Discretionary        0.33333              0.5            -0.16667
      2       Consumer Staples              0.66667              0.5             0.16667
      3       Consumer Discretionary        0.33333              0.5            -0.16667
      3       Consumer Staples              0.66667              0.5             0.16667
```

Input Arguments

`brinsonAttributionObj` — Brinson attribution model
object

Brinson attribution model, specified as a `brinsonAttribution` object.

Data Types: `object`

Output Arguments

`AverageCategoryWeights` — Category weights averaged over all periods
table

Category weights averaged over all periods, returned as a table with the following columns:

- `Category` — Asset category
- `AveragePortfolioWeight` — Average portfolio weights
- `AverageBenchmarkWeight` — Average benchmark weights
- `AverageActiveWeight` — Average active weights

`PeriodicCategoryWeights` — Category weights for each period
table

Category weights for each period, returned as a table with the following columns:

- `Period` — Time period numbers (1 for the first period, 2 for the second period, and so on)

- `Category` — Asset category
- `PortfolioWeight` — Portfolio weights
- `BenchmarkWeight` — Benchmark weights
- `ActiveWeight` — Active weights

Version History

Introduced in R2022b

References

[1] Brinson, G. P. and Fachler, N. "Measuring Non-US Equity Portfolio Performance." *Journal of Portfolio Management.* Spring 1985: 73–76.

[2] Brinson, G. P., Hood, L. R., and Beebower, G. L. "Determinants of Portfolio Performance." *Financial Analysts Journal.* Vol. 42, No. 4, 1986: 39–44.

[3] Menchero, J. "Multiperiod Arithmetic Attribution." *Financial Analysts Journal.* Vol. 60, No. 4, 2004: 76–91.

[4] Tuttle, D. L., Pinto, J. E., and McLeavey, D. W. *Managing Investment Portfolios: A Dynamic Process.* Third Edition. CFA Institute, 2007.

See Also

`totalAttribution` | `categoryReturns` | `categoryAttribution` | `summary` | `categoryReturnsChart` | `categoryWeightsChart` | `attributesChart`

Topics

"Analyze Performance Attribution Using Brinson Model" on page 4-320
"Backtest with Brinson Attribution to Evaluate Portfolio Performance" on page 4-311

totalAttribution

Compute total performance attribution by Brinson model

Syntax

```
[AggregateTotalAttribution,PeriodicTotalAttribution] = totalAttribution(
brinsonAttributionObj)
```

Description

`[AggregateTotalAttribution,PeriodicTotalAttribution] = totalAttribution(brinsonAttributionObj)` computes the total performance attribution of the portfolio, summed up for all categories, using the Brinson model.

Examples

Compute Total Performance Attribution Using `brinsonAttribution` Object

This example shows how to create a `brinsonAttribution` object and then use `totalAttribution` to compute the total performance attribution of the portfolio summed up for all categories (sectors).

Prepare Data

Create a table for the monthly prices for four assets.

```
 GM =[17.82;22.68;19.37;20.28];
 HD = [39.79;39.12;40.67;40.96];
 KO = [38.98;39.44;40.00;40.20];
 PG = [56.38;57.08;57.76;55.54];
 MonthlyPrices  = table(GM,HD,KO,PG);
```

Use `tick2ret` to define the monthly returns.

```
MonthlyReturns = tick2ret(MonthlyPrices.Variables)';
  [NumAssets,NumPeriods] = size(MonthlyReturns);
```

Define the periods.

```
Period = ones(NumAssets*NumPeriods,1);
  for k = 1:NumPeriods
      Period(k*NumAssets+1:end,1) = Period(k*NumAssets,1) + 1;
  end
```

Define the categories for the four assets.

```
Name = repmat(string(MonthlyPrices.Properties.VariableNames(:)),NumPeriods,1);
  Categories = repmat(categorical([ ...
      "Consumer Discretionary"; ...
      "Consumer Discretionary"; ...
      "Consumer Staples"; ...
      "Consumer Staples"]),NumPeriods,1);
```

Define benchmark and portfolio weights.

```
BenchmarkWeight = repmat(1./NumAssets.*ones(NumAssets, 1),NumPeriods,1);
  PortfolioWeight = repmat([1;0;1;1]./3,NumPeriods,1);
```

Create AssetTable Input

Create `AssetTable` as the input for the `brinsonAttribution` object.

```
  AssetTable = table(Period, Name, ...
      MonthlyReturns(:), Categories, PortfolioWeight, BenchmarkWeight, ...
      VariableNames=["Period","Name","Return","Category","PortfolioWeight","BenchmarkWeight"])
```

```
AssetTable=12×6 table
    Period    Name     Return             Category            PortfolioWeight    BenchmarkWeight
    ______    ____    _________    ______________________    _______________    _______________

      1       "GM"      0.27273    Consumer Discretionary        0.33333             0.25
      1       "HD"    -0.016838    Consumer Discretionary              0             0.25
      1       "KO"     0.011801    Consumer Staples              0.33333             0.25
      1       "PG"     0.012416    Consumer Staples              0.33333             0.25
      2       "GM"     -0.14594    Consumer Discretionary        0.33333             0.25
      2       "HD"     0.039622    Consumer Discretionary              0             0.25
      2       "KO"     0.014199    Consumer Staples              0.33333             0.25
      2       "PG"     0.011913    Consumer Staples              0.33333             0.25
      3       "GM"      0.04698    Consumer Discretionary        0.33333             0.25
      3       "HD"    0.0071306    Consumer Discretionary              0             0.25
      3       "KO"        0.005    Consumer Staples              0.33333             0.25
      3       "PG"    -0.038435    Consumer Staples              0.33333             0.25
```

Create brinsonAttribution Object

Use `brinsonAttribution` to create the `brinsonAttribution` object.

```
  BrinsonPAobj = brinsonAttribution(AssetTable)
```

```
BrinsonPAobj = 
  brinsonAttribution with properties:

                NumAssets: 4
       NumPortfolioAssets: 3
       NumBenchmarkAssets: 4
               NumPeriods: 3
            NumCategories: 2
                AssetName: [4x1 string]
              AssetReturn: [4x3 double]
            AssetCategory: [4x3 categorical]
     PortfolioAssetWeight: [4x3 double]
     BenchmarkAssetWeight: [4x3 double]
  PortfolioCategoryReturn: [2x3 double]
  BenchmarkCategoryReturn: [2x3 double]
  PortfolioCategoryWeight: [2x3 double]
  BenchmarkCategoryWeight: [2x3 double]
          PortfolioReturn: 0.0598
          BenchmarkReturn: 0.0540
```

```
            ActiveReturn: 0.0059
```

Compute Total Performance Attribution

Use the brinsonAttribution object with `totalAttribution` to compute the total performance attribution of the portfolio summed up for all categories.

```
[AggregateTotalAttribution, PeriodicTotalAttribution] = totalAttribution(BrinsonPAobj)
```

```
AggregateTotalAttribution=1×4 table
    Allocation    Selection    Interaction    ActiveReturn
    __________    _________    ___________    ____________

    -0.014353     0.030316      -0.010105      0.0058578  


PeriodicTotalAttribution=3×6 table
    Period    LinkingCoefficient    Allocation    Selection    Interaction    ActiveReturn
    ______    __________________    __________    _________    ___________    ____________

      1            0.97032           -0.019306     0.072391      -0.02413        0.028955 
      2             1.0838            0.011036    -0.046391      0.015464       -0.019891 
      3             1.0391          -0.0072954    0.0099623    -0.0033208     -0.00065389 
```

Input Arguments

`brinsonAttributionObj` — Brinson attribution model

object

Brinson attribution model, specified as a `brinsonAttribution` object.

Data Types: `object`

Output Arguments

`AggregateTotalAttribution` — Total attribution aggregated over all periods

table

Total attribution aggregated over all periods, returned as a table with the following columns:

- `Allocation` — Category allocation effect for the category
- `Selection` — Effect of selecting individual assets within the category
- `Interaction` — Allocation-selection interaction effect
- `ActiveReturn` — Active return compared with benchmark

`PeriodicTotalAttribution` — Total attribution for each period

table

Total attribution for each period, returned as a table with the following columns:

- `Period` — Time period numbers (1 for the first period, 2 for the second period, and so on)
- `LinkingCoefficient` — Linking coefficients used to aggregate the attribution results over all time periods

- `Allocation` — Category allocation effect for the category
- `Selection` — Effect of selecting individual assets within the category
- `Interaction` — Allocation-selection interaction effect
- `ActiveReturn` — Active return compared with benchmark

Version History

Introduced in R2022b

References

[1] Brinson, G. P. and Fachler, N. “Measuring Non-US Equity Portfolio Performance.” *Journal of Portfolio Management.* Spring 1985: 73–76.

[2] Brinson, G. P., Hood, L. R., and Beebower, G. L. “Determinants of Portfolio Performance.” *Financial Analysts Journal.* Vol. 42, No. 4, 1986: 39–44.

[3] Menchero, J. “Multiperiod Arithmetic Attribution.” *Financial Analysts Journal.* Vol. 60, No. 4, 2004: 76–91.

[4] Tuttle, D. L., Pinto, J. E., and McLeavey, D. W. *Managing Investment Portfolios: A Dynamic Process.* Third Edition. CFA Institute, 2007.

See Also

`categoryWeights` | `categoryReturns` | `categoryAttribution` | `summary` | `categoryReturnsChart` | `categoryWeightsChart` | `attributionsChart`

Topics

“Analyze Performance Attribution Using Brinson Model” on page 4-320
“Backtest with Brinson Attribution to Evaluate Portfolio Performance” on page 4-311

summary

Summarize performance attribution by Brinson model

Syntax

```
SummaryTable = summary(brinsonAttributionObj)
```

Description

`SummaryTable = summary(brinsonAttributionObj)` generates a table that summarizes the final results (aggregated over all time periods and categories) of the performance attribution by the Brinson model.

Examples

Generate Summary Table Using `brinsonAttribution` Object

This example shows how to create a `brinsonAttribution` object and then use `summary` to generate a table that summarizes the final results of the performance attribution by the Brinson model, aggregated over all time periods and categories (sectors).

Prepare Data

Create a table for the monthly prices for four assets.

```
 GM =[17.82;22.68;19.37;20.28];
 HD = [39.79;39.12;40.67;40.96];
 KO = [38.98;39.44;40.00;40.20];
 PG = [56.38;57.08;57.76;55.54];
 MonthlyPrices  = table(GM,HD,KO,PG);
```

Use `tick2ret` to define the monthly returns.

```
MonthlyReturns = tick2ret(MonthlyPrices.Variables)';
  [NumAssets,NumPeriods] = size(MonthlyReturns);
```

Define the periods.

```
Period = ones(NumAssets*NumPeriods,1);
  for k = 1:NumPeriods
      Period(k*NumAssets+1:end,1) = Period(k*NumAssets,1) + 1;
  end
```

Define the categories for the four assets.

```
Name = repmat(string(MonthlyPrices.Properties.VariableNames(:)),NumPeriods,1);
  Categories = repmat(categorical([ ...
      "Consumer Discretionary"; ...
      "Consumer Discretionary"; ...
      "Consumer Staples"; ...
      "Consumer Staples"]),NumPeriods,1);
```

Define benchmark and portfolio weights.

```
BenchmarkWeight = repmat(1./NumAssets.*ones(NumAssets, 1),NumPeriods,1);
  PortfolioWeight = repmat([1;0;1;1]./3,NumPeriods,1);
```

Create AssetTable Input

Create AssetTable as the input for the brinsonAttribution object.

```
  AssetTable = table(Period, Name, ...
      MonthlyReturns(:), Categories, PortfolioWeight, BenchmarkWeight, ...
      VariableNames=["Period","Name","Return","Category","PortfolioWeight","BenchmarkWeight"])
```

```
AssetTable=12×6 table
    Period    Name      Return              Category           PortfolioWeight    BenchmarkWeight
    ______    ____    _________    ______________________    _______________    _______________

      1       "GM"      0.27273    Consumer Discretionary        0.33333              0.25
      1       "HD"    -0.016838    Consumer Discretionary              0              0.25
      1       "KO"     0.011801    Consumer Staples              0.33333              0.25
      1       "PG"     0.012416    Consumer Staples              0.33333              0.25
      2       "GM"     -0.14594    Consumer Discretionary        0.33333              0.25
      2       "HD"     0.039622    Consumer Discretionary              0              0.25
      2       "KO"     0.014199    Consumer Staples              0.33333              0.25
      2       "PG"     0.011913    Consumer Staples              0.33333              0.25
      3       "GM"      0.04698    Consumer Discretionary        0.33333              0.25
      3       "HD"    0.0071306    Consumer Discretionary              0              0.25
      3       "KO"        0.005    Consumer Staples              0.33333              0.25
      3       "PG"    -0.038435    Consumer Staples              0.33333              0.25
```

Create brinsonAttribution Object

Use brinsonAttribution to create the brinsonAttribution object.

```
  BrinsonPAobj = brinsonAttribution(AssetTable)
```

```
BrinsonPAobj = 
  brinsonAttribution with properties:

                NumAssets: 4
       NumPortfolioAssets: 3
       NumBenchmarkAssets: 4
               NumPeriods: 3
            NumCategories: 2
                AssetName: [4x1 string]
              AssetReturn: [4x3 double]
            AssetCategory: [4x3 categorical]
     PortfolioAssetWeight: [4x3 double]
     BenchmarkAssetWeight: [4x3 double]
  PortfolioCategoryReturn: [2x3 double]
  BenchmarkCategoryReturn: [2x3 double]
  PortfolioCategoryWeight: [2x3 double]
  BenchmarkCategoryWeight: [2x3 double]
          PortfolioReturn: 0.0598
          BenchmarkReturn: 0.0540
             ActiveReturn: 0.0059
```

Generate Summary Table

Use the brinsonAttribution object with `summary` to generate a table that summarizes the final results of the performance attribution by the Brinson model, aggregated over all time periods and categories.

```
SummaryTable = summary(BrinsonPAobj)
```

```
SummaryTable=11×1 table
                                     Brinson Attribution Summary
                                     ___________________________

    Total Number of Assets                        4
    Number of Assets in Portfolio                 3
    Number of Assets in Benchmark                 4
    Number of Periods                             3
    Number of Categories                          2
    Portfolio Return                       0.059847
    Benchmark Return                        0.05399
    Active Return                         0.0058578
    Allocation Effect                     -0.014353
    Selection Effect                       0.030316
    Interaction Effect                    -0.010105
```

Input Arguments

`brinsonAttributionObj` — Brinson attribution model
object

Brinson attribution model, specified as a `brinsonAttribution` object.

Data Types: `object`

Output Arguments

`SummaryTable` — Metrics summarizing performance attribution by Brinson model
table

Metrics summarizing the performance attribution by the Brinson model, returned as a table where each row of the table is a calculated metric as follows:

- `Total Number of Assets` — Total for assets in the `AssetTable`
- `Number of Assets in Portfolio` — Total assets in the portfolio, as defined in `AssetTable`
- `Number of Assets in Benchmark` — The total assets in the benchmark, as defined in `AssetTable`
- `Number of Periods` — Total number of `Periods`, as defined in `AssetTable`
- `Portfolio Return` — Portfolio return, as defined in `AssetTable`
- `Benchmark Return` — Benchmark return, as defined in `AssetTable`
- `Active Return` — Calculated active return for the portfolio, as defined in `AssetTable`
- `Allocation Effect` — Allocation effect for the portfolio, as defined in `AssetTable`
- `Selection Effect` — Selection effect for the portfolio, as defined in `AssetTable`

- `Interaction Effect` — Interaction effect for the portfolio, as defined in `AssetTable`

Version History

Introduced in R2022b

References

[1] Brinson, G. P. and Fachler, N. “Measuring Non-US Equity Portfolio Performance.” *Journal of Portfolio Management.* Spring 1985: 73–76.

[2] Brinson, G. P., Hood, L. R., and Beebower, G. L. “Determinants of Portfolio Performance.” *Financial Analysts Journal.* Vol. 42, No. 4, 1986: 39–44.

[3] Menchero, J. “Multiperiod Arithmetic Attribution.” *Financial Analysts Journal.* Vol. 60, No. 4, 2004: 76–91.

[4] Tuttle, D. L., Pinto, J. E., and McLeavey, D. W. *Managing Investment Portfolios: A Dynamic Process.* Third Edition. CFA Institute, 2007.

See Also

`totalAttribution` | `categoryWeights` | `categoryReturns` | `categoryAttribution` | `categoryReturnsChart` | `categoryWeightsChart` | `attributionsChart`

Topics

“Analyze Performance Attribution Using Brinson Model” on page 4-320
“Backtest with Brinson Attribution to Evaluate Portfolio Performance” on page 4-311

attributionsChart

Create horizontal bar chart of performance attribution

Syntax

```
attributionsChart(BrinsonPAObj)
h = attributionsChart(ax,BrinsonPAObj)
h = attributionsChart( ___ ,Name=Value)
```

Description

`attributionsChart(BrinsonPAObj)` creates a horizontal bar chart of portfolio performance attributions by category, aggregated over all time periods using a `brinsonAttribution` object. This function decomposes active returns into category allocation, within-category selection, and allocation-selection interaction effects.

`h = attributionsChart(ax,BrinsonPAObj)` additionally returns the figure handle `h`.

`h = attributionsChart( ___ ,Name=Value)` adds optional name-value arguments.

Examples

Create Horizontal Bar Chart of Performance Attribution

This example shows how to create a `brinsonAttribution` object that you can then use with the `attributionsChart` function to generate a bar chart of performance attribution.

Prepare Data

Create a table for the monthly prices for four assets.

```
GM =[17.82;22.68;19.37;20.28];
HD = [39.79;39.12;40.67;40.96];
KO = [38.98;39.44;40.00;40.20];
PG = [56.38;57.08;57.76;55.54];
MonthlyPrices  = table(GM,HD,KO,PG);
```

Use `tick2ret` to define the monthly returns.

```
MonthlyReturns = tick2ret(MonthlyPrices.Variables)';
  [NumAssets,NumPeriods] = size(MonthlyReturns);
```

Define the periods.

```
Period = ones(NumAssets*NumPeriods,1);
  for k = 1:NumPeriods
      Period(k*NumAssets+1:end,1) = Period(k*NumAssets,1) + 1;
  end
```

Define the categories (sectors) for the four assets.

```
Name = repmat(string(MonthlyPrices.Properties.VariableNames(:)),NumPeriods,1);
  Categories = repmat(categorical([ ...
      "Consumer Discretionary"; ...
      "Consumer Discretionary"; ...
      "Consumer Staples"; ...
      "Consumer Staples"]),NumPeriods,1);
```

Define benchmark and portfolio weights.

```
BenchmarkWeight = repmat(1./NumAssets.*ones(NumAssets,1),NumPeriods,1);
  PortfolioWeight = repmat([1;0;1;1]./3,NumPeriods,1);
```

Create AssetTable Input

Create `AssetTable` as the input for the `brinsonAttribution` object.

```
AssetTable = table(Period, Name, ...
   MonthlyReturns(:), Categories, PortfolioWeight, BenchmarkWeight, ...
   VariableNames=["Period","Name","Return","Category","PortfolioWeight","BenchmarkWeight"])
```

```
AssetTable=12×6 table
    Period     Name      Return              Category            PortfolioWeight    BenchmarkWeight
    ______     ____    _________    ______________________    _______________    _______________

      1        "GM"      0.27273    Consumer Discretionary        0.33333             0.25      
      1        "HD"    -0.016838    Consumer Discretionary              0             0.25      
      1        "KO"     0.011801    Consumer Staples              0.33333             0.25      
      1        "PG"     0.012416    Consumer Staples              0.33333             0.25      
      2        "GM"     -0.14594    Consumer Discretionary        0.33333             0.25      
      2        "HD"     0.039622    Consumer Discretionary              0             0.25      
      2        "KO"     0.014199    Consumer Staples              0.33333             0.25      
      2        "PG"     0.011913    Consumer Staples              0.33333             0.25      
      3        "GM"      0.04698    Consumer Discretionary        0.33333             0.25      
      3        "HD"    0.0071306    Consumer Discretionary              0             0.25      
      3        "KO"        0.005    Consumer Staples              0.33333             0.25      
      3        "PG"    -0.038435    Consumer Staples              0.33333             0.25      
```

Create brinsonAttribution Object

Use `brinsonAttribution` to create the `brinsonAttribution` object.

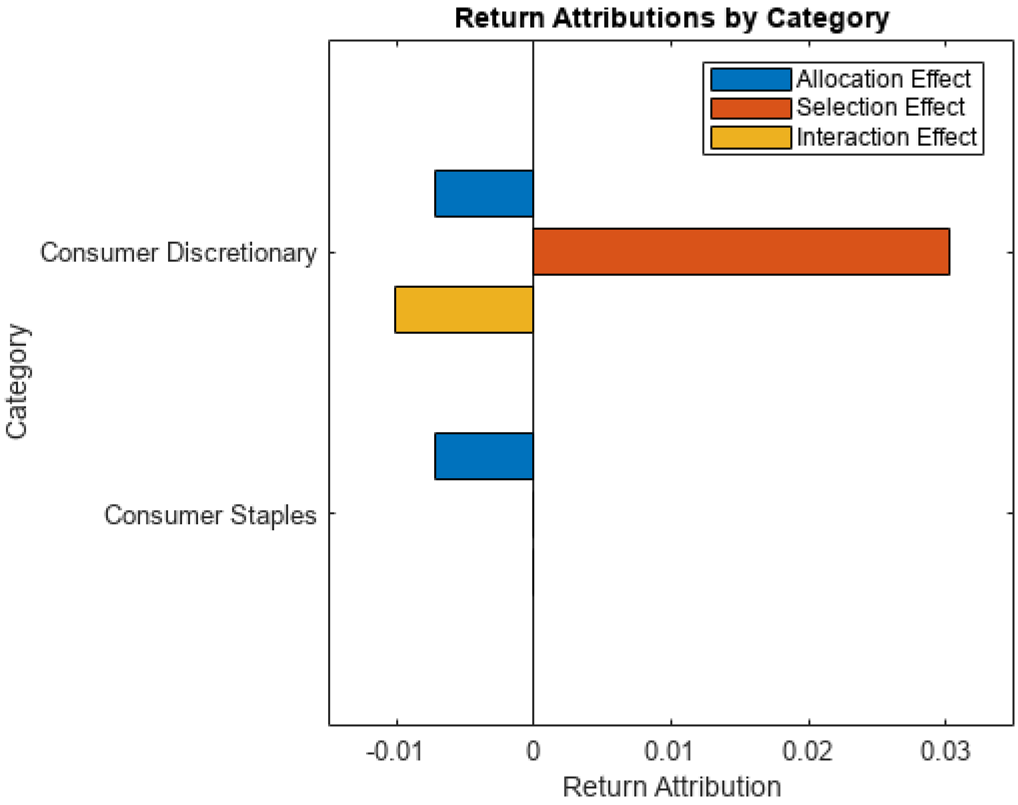
```
BrinsonPAobj = brinsonAttribution(AssetTable)
```

```
BrinsonPAobj = 
  brinsonAttribution with properties:

                  NumAssets: 4
         NumPortfolioAssets: 3
         NumBenchmarkAssets: 4
                 NumPeriods: 3
              NumCategories: 2
                  AssetName: [4x1 string]
                AssetReturn: [4x3 double]
              AssetCategory: [4x3 categorical]
       PortfolioAssetWeight: [4x3 double]
       BenchmarkAssetWeight: [4x3 double]
    PortfolioCategoryReturn: [2x3 double]
    BenchmarkCategoryReturn: [2x3 double]
    PortfolioCategoryWeight: [2x3 double]
    BenchmarkCategoryWeight: [2x3 double]
            PortfolioReturn: 0.0598
            BenchmarkReturn: 0.0540
               ActiveReturn: 0.0059
```

Generate Horizontal Bar Chart for Performance Attribution

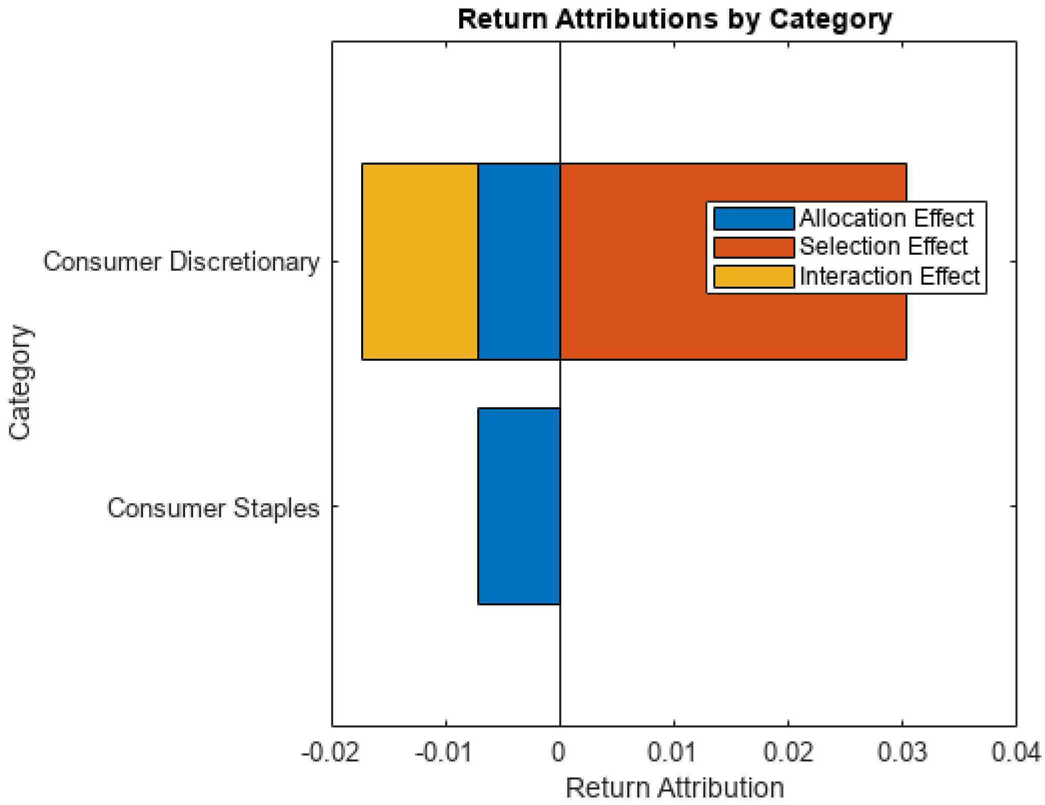
Use the `brinsonAttribution` object with `attributionsChart` to generate a horizontal bar chart of portfolio performance attributions by category, aggregated over all time periods.

```
attributionsChart(BrinsonPAobj)
```



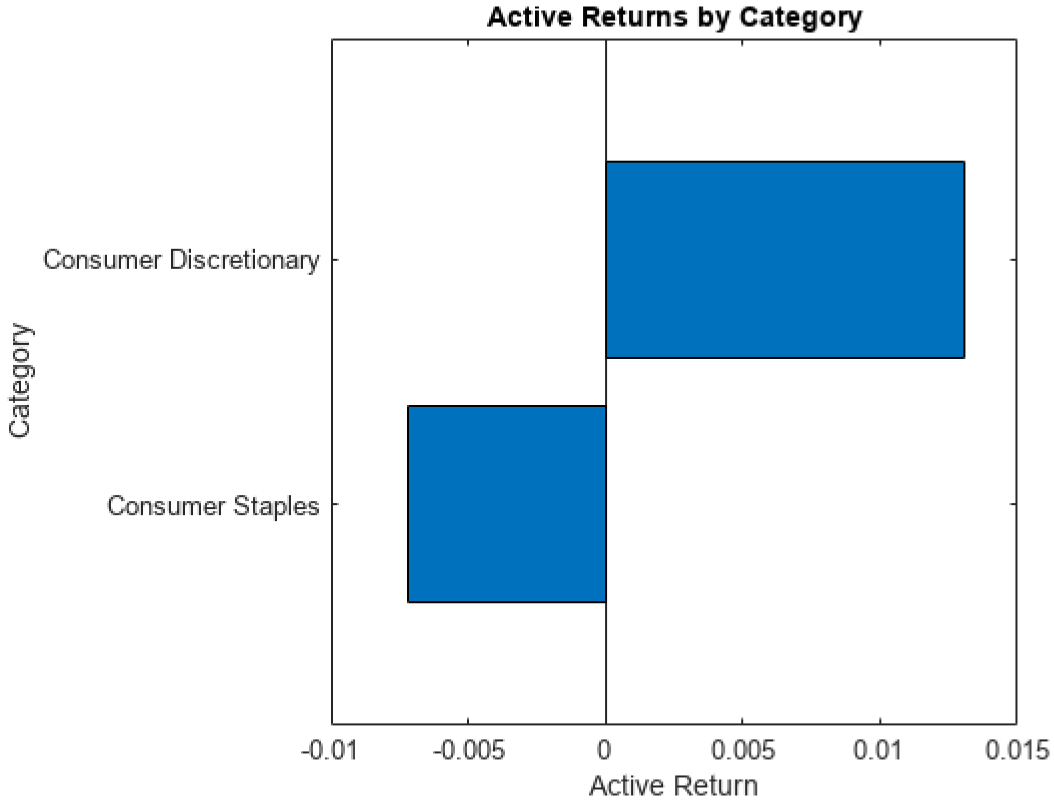
Alternatively, you can use the name-value argument for `Style` to generate a stacked horizontal bar chart of return attributions by category.

```
attributionsChart(BrinsonPAobj,Style="stacked")
```



Also, you can use the name-value argument for `Style` to generate a horizontal bar chart of active returns by category.

```
attributionsChart(BrinsonPAobj,Style="active")
```



Input Arguments

`BrinsonPAObj` — `brinsonAttribution` object to analyze performance attribution
`brinsonAttribution` object

`brinsonAttribution` object to analyze performance attribution. Use `brinsonAttribution` to create the `brinsonAttribution` object.

Data Types: `object`

`ax` — Valid axis object
`ax` object

(Optional) Valid axis object, specified as an `ax` object that you create using `axes`. `attributesChart` creates the plot on the axes specified by the optional `ax` argument instead of on the current axes (gca). The optional argument `ax` can precede any of the input argument combinations. If you do not specify an `axes` object, `attributesChart` plots into the current axes.

Data Types: `object`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `attributionsChart(BrinsonPAObj,Style="stacked")`

`Style` — Indicator to select style of attribution bar graphs

`"grouped"` (default) | character vector with value `'grouped'`, `'stacked'`, or `'active'` | string with value `"grouped"`, `"stacked"`, or `"active"`

Indicator to select style of attribution bar graphs, specified as `Style` and one of the following values:

- `"grouped"` — Plot the bar graphs for the allocation, selection, and interaction effects side-by-side.
- `"stacked"` — Stack the bar graphs for the allocation, selection, and interaction effects.
- `"active"` — Plot the bar graphs for the active returns as the sum of allocation, selection, and interaction effects.

Data Types: `char` | `string`

Output Arguments

`h` — Figure handle

handle object

Figure handle for the performance attributions chart, returned as handle object. You can use the figure handle to access and change the properties of the chart.

Version History

Introduced in R2023a

See Also

`categoryReturnsChart` | `categoryWeightsChart` | `brinsonAttribution`

Topics

“Analyze Performance Attribution Using Brinson Model” on page 4-320
“Backtest with Brinson Attribution to Evaluate Portfolio Performance” on page 4-311

categoryReturnsChart

Create horizontal bar chart of category returns

Syntax

```
categoryReturnsChart(BrinsonPAObj)
h = categoryReturnsChart(ax,BrinsonPAObj)
```

Description

`categoryReturnsChart(BrinsonPAObj)` creates a horizontal bar chart of portfolio and benchmark category returns, aggregated over all time periods using a `brinsonAttribution` object.

`h = categoryReturnsChart(ax,BrinsonPAObj)` additionally returns the figure handle `h`.

Examples

Create Horizontal Bar Chart of Category Returns

This example shows how to create a `brinsonAttribution` object that you can then use with the `categoryReturnsChart` function to generate a bar chart of category returns.

Prepare Data

Create a table for the monthly prices for four assets.

```
 GM =[17.82;22.68;19.37;20.28];
 HD = [39.79;39.12;40.67;40.96];
 KO = [38.98;39.44;40.00;40.20];
 PG = [56.38;57.08;57.76;55.54];
 MonthlyPrices  = table(GM,HD,KO,PG);
```

Use `tick2ret` to define the monthly returns.

```
MonthlyReturns = tick2ret(MonthlyPrices.Variables)';
  [NumAssets,NumPeriods] = size(MonthlyReturns);
```

Define the periods.

```
Period = ones(NumAssets*NumPeriods,1);
  for k = 1:NumPeriods
      Period(k*NumAssets+1:end,1) = Period(k*NumAssets,1) + 1;
  end
```

Define the categories (sectors) for the four assets.

```
Name = repmat(string(MonthlyPrices.Properties.VariableNames(:)),NumPeriods,1);
  Categories = repmat(categorical([ ...
      "Consumer Discretionary"; ...
      "Consumer Discretionary"; ...
      "Consumer Staples"; ...
      "Consumer Staples"]),NumPeriods,1);
```

Define benchmark and portfolio weights.

```
BenchmarkWeight = repmat(1./NumAssets.*ones(NumAssets, 1),NumPeriods,1);
  PortfolioWeight = repmat([1;0;1;1]./3,NumPeriods,1);
```

Create AssetTable Input

Create AssetTable as the input for the brinsonAttribution object.

```
  AssetTable = table(Period, Name, ...
      MonthlyReturns(:), Categories, PortfolioWeight, BenchmarkWeight, ...
      VariableNames=["Period","Name","Return","Category","PortfolioWeight","BenchmarkWeight"])
```

```
AssetTable=12×6 table
    Period    Name      Return             Category            PortfolioWeight    BenchmarkWeight
    ______    ____    _________    ______________________    _______________    _______________

      1       "GM"      0.27273    Consumer Discretionary        0.33333              0.25
      1       "HD"    -0.016838    Consumer Discretionary              0              0.25
      1       "KO"     0.011801    Consumer Staples              0.33333              0.25
      1       "PG"     0.012416    Consumer Staples              0.33333              0.25
      2       "GM"     -0.14594    Consumer Discretionary        0.33333              0.25
      2       "HD"     0.039622    Consumer Discretionary              0              0.25
      2       "KO"     0.014199    Consumer Staples              0.33333              0.25
      2       "PG"     0.011913    Consumer Staples              0.33333              0.25
      3       "GM"      0.04698    Consumer Discretionary        0.33333              0.25
      3       "HD"    0.0071306    Consumer Discretionary              0              0.25
      3       "KO"        0.005    Consumer Staples              0.33333              0.25
      3       "PG"    -0.038435    Consumer Staples              0.33333              0.25
```

Create brinsonAttribution Object

Use brinsonAttribution to create the brinsonAttribution object.

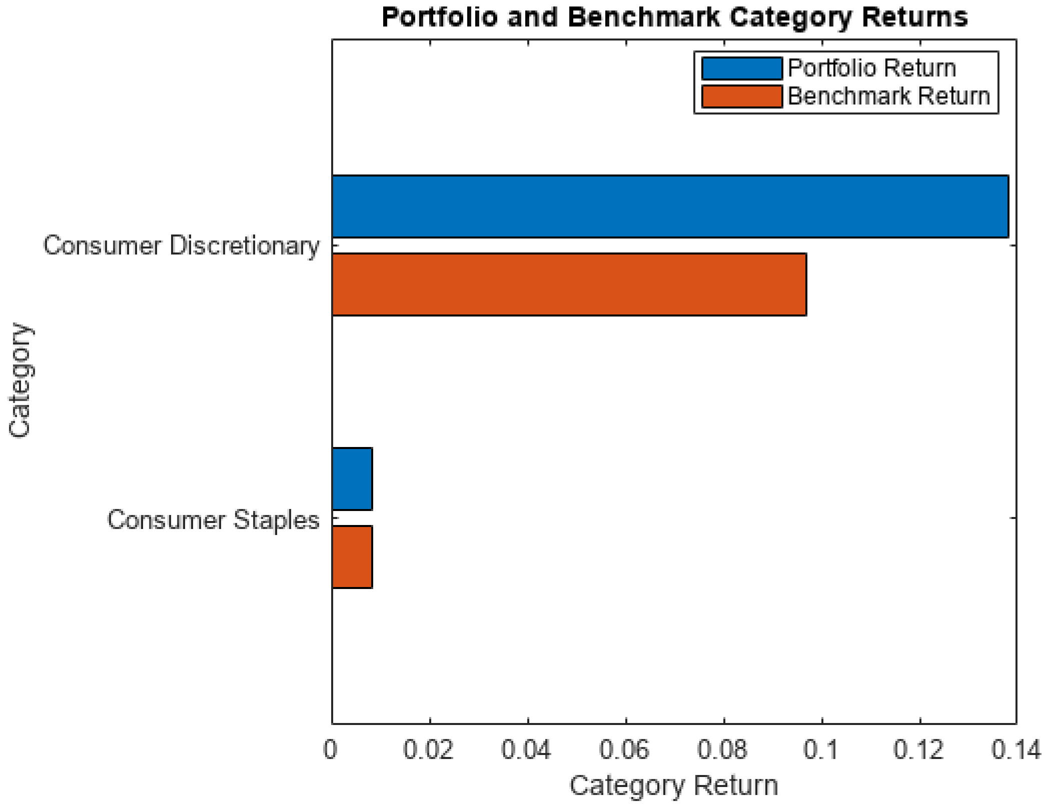
```
  BrinsonPAobj = brinsonAttribution(AssetTable)
```

```
BrinsonPAobj = 
  brinsonAttribution with properties:

                NumAssets: 4
       NumPortfolioAssets: 3
       NumBenchmarkAssets: 4
               NumPeriods: 3
            NumCategories: 2
                AssetName: [4x1 string]
              AssetReturn: [4x3 double]
            AssetCategory: [4x3 categorical]
     PortfolioAssetWeight: [4x3 double]
     BenchmarkAssetWeight: [4x3 double]
  PortfolioCategoryReturn: [2x3 double]
  BenchmarkCategoryReturn: [2x3 double]
  PortfolioCategoryWeight: [2x3 double]
  BenchmarkCategoryWeight: [2x3 double]
          PortfolioReturn: 0.0598
          BenchmarkReturn: 0.0540
             ActiveReturn: 0.0059
```

Generate Horizontal Bar Chart for Category Returns

Use the `brinsonAttribution` object with `categoryReturnsChart` to generate a horizontal bar chart of portfolio and benchmark category returns, aggregated over all time periods.

```
categoryReturnsChart(BrinsonPAobj)
```



Input Arguments

`BrinsonPAObj` — `brinsonAttribution` object to analyze performance attribution
`brinsonAttribution` object

`brinsonAttribution` object to analyze performance attribution. Use `brinsonAttribution` to create the `brinsonAttribution` object.

Data Types: `object`

`ax` — Valid axis object
`ax` object

(Optional) Valid axis object, specified as an `ax` object that you create using `axes`. `categoryReturnsChart` creates the plot on the axes specified by the optional `ax` argument instead of on the current axes (gca). The optional argument `ax` can precede any of the input argument combinations. If you do not specify an `axes` object, `categoryReturnsChart` plots into the current axes.

Data Types: `object`

Output Arguments

h — Figure handle
handle object

Figure handle for the category returns chart, returned as handle object. You can use the figure handle to access and change the properties of the chart.

Version History

Introduced in R2023a

See Also

`categoryWeightsChart` | `attributionsChart` | `brinsonAttribution`

Topics

“Analyze Performance Attribution Using Brinson Model” on page 4-320
“Backtest with Brinson Attribution to Evaluate Portfolio Performance” on page 4-311

categoryWeightsChart

Create a horizontal bar chart for category weights

Syntax

```
categoryWeightsChart(BrinsonPAObj)
h = categoryWeightsChart(ax,BrinsonPAObj)
h = categoryWeightsChart(___,Name=Value)
```

Description

`categoryWeightsChart(BrinsonPAObj)` creates a horizontal bar chart of portfolio, benchmark, and active weights by category, averaged over all time periods using a `brinsonAttribution` object.

`h = categoryWeightsChart(ax,BrinsonPAObj)` additionally returns the figure handle `h`.

`h = categoryWeightsChart(___,Name=Value)` adds optional name-value arguments.

Examples

Create Horizontal Bar Chart of Category Weights

This example shows how to create a `brinsonAttribution` object that you can then use with the `categoryWeightsChart` function to generate a bar chart of category weights.

Prepare Data

Create a table for the monthly prices for four assets.

```
GM =[17.82;22.68;19.37;20.28];
HD = [39.79;39.12;40.67;40.96];
KO = [38.98;39.44;40.00;40.20];
PG = [56.38;57.08;57.76;55.54];
MonthlyPrices  = table(GM,HD,KO,PG);
```

Use `tick2ret` to define the monthly returns.

```
MonthlyReturns = tick2ret(MonthlyPrices.Variables)';
  [NumAssets,NumPeriods] = size(MonthlyReturns);
```

Define the periods.

```
Period = ones(NumAssets*NumPeriods,1);
  for k = 1:NumPeriods
      Period(k*NumAssets+1:end,1) = Period(k*NumAssets,1) + 1;
  end
```

Define the categories (sectors) for the four assets.

```
Name = repmat(string(MonthlyPrices.Properties.VariableNames(:)),NumPeriods,1);
  Categories = repmat(categorical([ ...
      "Consumer Discretionary"; ...
      "Consumer Discretionary"; ...
      "Consumer Staples"; ...
      "Consumer Staples"]),NumPeriods,1);
```

Define benchmark and portfolio weights.

```
BenchmarkWeight = repmat(1./NumAssets.*ones(NumAssets, 1),NumPeriods,1);
PortfolioWeight = repmat([1;0;1;1]./3,NumPeriods,1);
```

Create AssetTable Input

Create `AssetTable` as the input for the `brinsonAttribution` object.

```
AssetTable = table(Period, Name, ...
    MonthlyReturns(:), Categories, PortfolioWeight, BenchmarkWeight, ...
    VariableNames=["Period","Name","Return","Category","PortfolioWeight","BenchmarkWeight"])
```

```
AssetTable=12×6 table
    Period    Name      Return             Category             PortfolioWeight    BenchmarkWeight
    ______    ____    _________    ______________________    _______________    _______________

      1       "GM"      0.27273    Consumer Discretionary        0.33333             0.25      
      1       "HD"    -0.016838    Consumer Discretionary              0             0.25      
      1       "KO"     0.011801    Consumer Staples              0.33333             0.25      
      1       "PG"     0.012416    Consumer Staples              0.33333             0.25      
      2       "GM"     -0.14594    Consumer Discretionary        0.33333             0.25      
      2       "HD"     0.039622    Consumer Discretionary              0             0.25      
      2       "KO"     0.014199    Consumer Staples              0.33333             0.25      
      2       "PG"     0.011913    Consumer Staples              0.33333             0.25      
      3       "GM"      0.04698    Consumer Discretionary        0.33333             0.25      
      3       "HD"    0.0071306    Consumer Discretionary              0             0.25      
      3       "KO"        0.005    Consumer Staples              0.33333             0.25      
      3       "PG"    -0.038435    Consumer Staples              0.33333             0.25      
```

Create brinsonAttribution Object

Use `brinsonAttribution` to create the `brinsonAttribution` object.

```
BrinsonPAobj = brinsonAttribution(AssetTable)
```

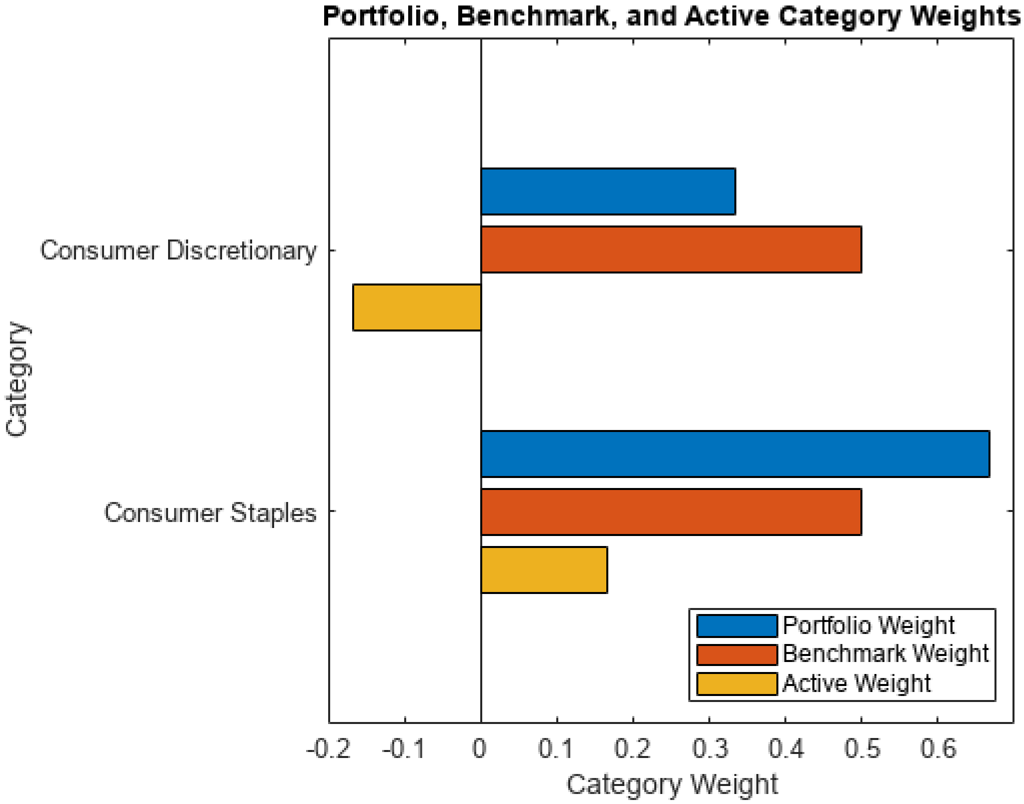
```
BrinsonPAobj = 
  brinsonAttribution with properties:

                  NumAssets: 4
         NumPortfolioAssets: 3
         NumBenchmarkAssets: 4
                 NumPeriods: 3
              NumCategories: 2
                  AssetName: [4x1 string]
                AssetReturn: [4x3 double]
              AssetCategory: [4x3 categorical]
       PortfolioAssetWeight: [4x3 double]
       BenchmarkAssetWeight: [4x3 double]
    PortfolioCategoryReturn: [2x3 double]
    BenchmarkCategoryReturn: [2x3 double]
    PortfolioCategoryWeight: [2x3 double]
    BenchmarkCategoryWeight: [2x3 double]
```

```
        PortfolioReturn: 0.0598
        BenchmarkReturn: 0.0540
           ActiveReturn: 0.0059
```

Generate Horizontal Bar Chart for Category Weights

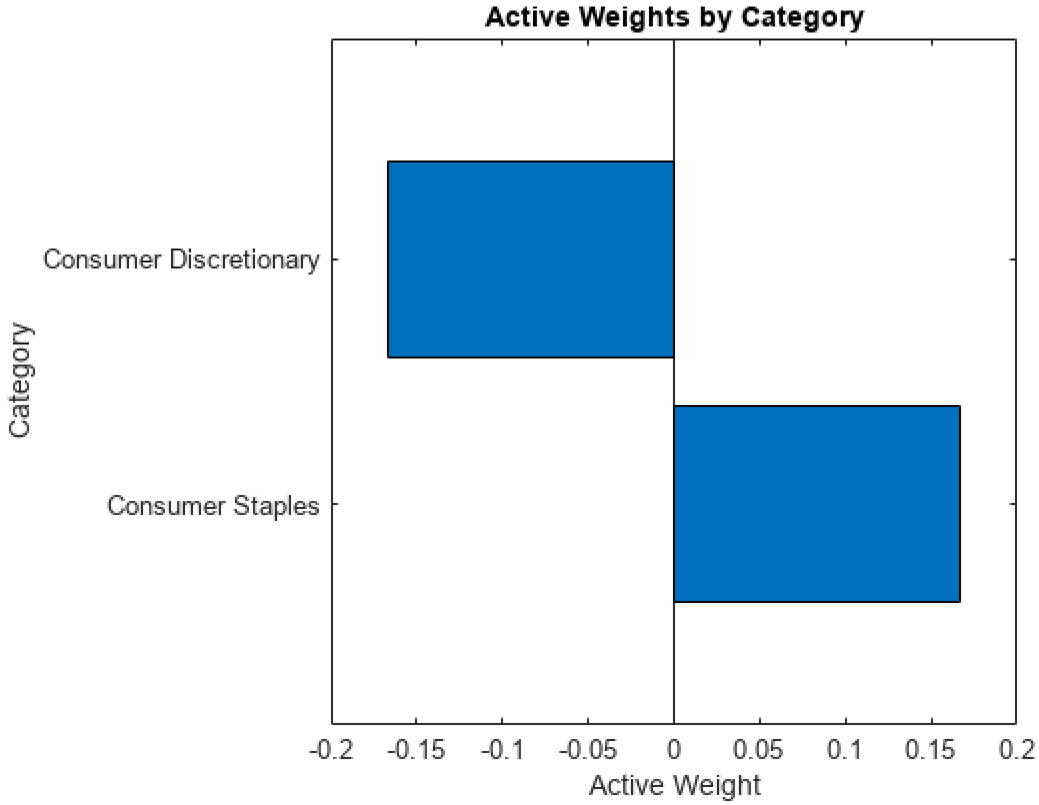
Use the `brinsonAttribution` object with `categoryWeightsChart` to generate a horizontal bar chart of portfolio, benchmark, and active weights by category, averaged over all time periods.

```
categoryWeightsChart(BrinsonPAobj)
```



Alternatively, you can use the name-value argument for `ActiveOnly` to plot only active weights by category.

```
categoryWeightsChart(BrinsonPAobj,ActiveOnly=true)
```



Input Arguments

`BrinsonPAObj` — `brinsonAttribution` object to analyze performance attribution
`brinsonAttribution` object

`brinsonAttribution` object to analyze performance attribution. Use `brinsonAttribution` to create the `brinsonAttribution` object.

Data Types: `object`

`ax` — Valid axis object
`ax` object

(Optional) Valid axis object, specified as an `ax` object that you create using `axes`. `categoryWeightsChart` creates the plot on the axes specified by the optional `ax` argument instead of on the current axes (gca). The optional argument `ax` can precede any of the input argument combinations. If you do not specify an `axes`, `categoryWeightsChart` plots into the current axes.

Data Types: `object`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `categoryWeightsChart(BrinsonPAObj,ActiveOnly=true)`

`ActiveOnly` — Indicator to plot only active weights by category

`false` (default) | logical with value `true` or `false`

Indicator to plot only active weights by category, specified as `ActiveOnly` and a logical with a value of `true` (plot active weights only) or `false` (plot portfolio, benchmark, and active weights).

Data Types: `logical`

Output Arguments

`h` — Figure handle

handle object

Figure handle for weights by category chart, returned as handle object. You can use the figure handle to access and change the properties of the chart.

Version History

Introduced in R2023a

See Also

`categoryReturnsChart` | `attributionsChart` | `brinsonAttribution`

Topics

“Analyze Performance Attribution Using Brinson Model” on page 4-320
“Backtest with Brinson Attribution to Evaluate Portfolio Performance” on page 4-311

covarianceDenoising

Estimate covariance matrix using denoising

Syntax

```
SigmaHat = covarianceDenoising(AssetReturns)
SigmaHat = covarianceDenoising(Sigma,SampleSize)
[SigmaHat,numSignalEig] = covarianceDenoising( ___ )
```

Description

`SigmaHat = covarianceDenoising(AssetReturns)` returns a covariance estimate that uses random matrix theory to denoise the empirical covariance matrix. For more information, see "Covariance Denoising" on page 15-55 and "Denoising Algorithm" on page 15-55.

In addition, you can use `covarianceShrinkage` to compute an estimate of covariance matrix using shrinkage estimation. For information on which covariance estimation method to choose, see "Comparison of Methods for Covariance Estimation" on page 4-134.

`SigmaHat = covarianceDenoising(Sigma,SampleSize)` returns a covariance estimate from an initial covariance matrix estimate (`Sigma`) and the sample size used to estimate the initial covariance (`sampleSize`).

`[SigmaHat,numSignalEig] = covarianceDenoising( ___ )` returns a covariance estimate and the number of eigenvalues that are associated with signal in combination with either of the input argument combinations in the previous syntaxes.

Examples

Perform Covariance Denoising

This example shows how to use covariance denoising to reduce noise and enhance the signal of the empirical covariance matrix. In mean-variance portfolio optimization, a noisy estimate of the covariance estimate results in unstable solutions that cause high turnover and transaction costs. Ideally, to decrease the estimation error, it is desirable to increase the sample size. Yet, there are cases where this is not possible. In extreme cases in which the number of assets is larger than the number of observations, the traditional covariance matrix results in a singular matrix. Working with a nearly singular or an ill-conditioned covariance matrix magnifies the impact of estimation errors.

Compute a portfolio efficient frontier using different covariance estimates with the same sample data.

```
% Load portfolio data with 225 assets
load port5.mat
covariance = corr2cov(stdDev_return,Correlation);
% Generate a sample with 200 observations
rng('default')
nScen = 200;
retSeries = portsim(mean_return',covariance,nScen);
```

Compute the traditional and denoised covariance estimates. Use `covarianceDenoising` to create a covariance estimate.

```
Sigma = cov(retSeries);
denoisedSigma = covarianceDenoising(retSeries);
```

Compute the condition number of both covariance estimates. The denoised covariance `denoisedSigma` has a lower condition number than the traditional covariance estimate `Sigma`.

```
conditionNum = [cond(Sigma); cond(denoisedSigma)];
condNumT = table(conditionNum,'RowNames',{'Sigma','SigmaHat'})
```

```
condNumT=2×1 table
                conditionNum
                ____________

    Sigma        2.5285e+18
    SigmaHat         1046.8
```

Use `Portfolio` to construct `Portfolio` objects that use the different `AssetCovar` values. Then use `setDefaultConstraints` to set the portfolio constraints with nonnegative weights that sum to 1 for the three portfolios: the true mean with the true covariance, the traditional covariance estimate, and the denoised estimate.

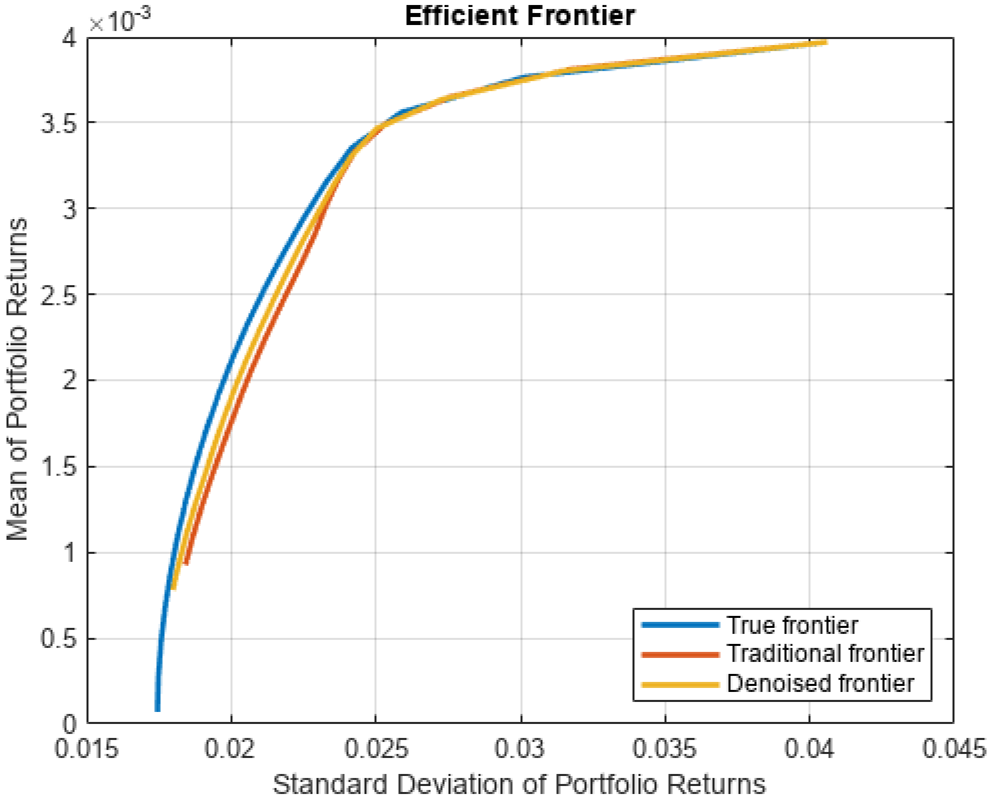
```
% Create a Portfolio object with the true parameters
p = Portfolio(AssetMean=mean_return,AssetCovar=covariance);
p = setDefaultConstraints(p);

% Create a Portfolio object with true mean and traditional covariance estimate
pTraditional = Portfolio(AssetMean=mean_return,AssetCovar=Sigma);
pTraditional = setDefaultConstraints(pTraditional);

% Create a Portfolio object with true mean and denoised covariance
pDenoised = Portfolio(AssetMean=mean_return,AssetCovar=denoisedSigma);
pDenoised = setDefaultConstraints(pDenoised);
```

Use `estimateFrontier` to estimate the efficient frontier for each of the `Portfolio` objects.

```
% Number of portfolios on the efficient frontier
nPort = 20;

% True efficient portfolios
w = estimateFrontier(p,nPort);

% Traditional covariance efficient portfolios
wTraditional = estimateFrontier(pTraditional,nPort);

% Denoised covariance efficient portfolios
wDenoised = estimateFrontier(pDenoised,nPort);
```

Use `plotFrontier` to plot the frontier obtained from the different weights using the true parameter values.

```
figure
plotFrontier(p,w)
hold on
plotFrontier(p,wTraditional)
plotFrontier(p,wDenoised)
legend('True frontier','Traditional frontier','Denoised frontier', ...
    Location='southeast');
hold off
```



The efficient frontier obtained by denoising is closer to the true frontier than the frontier obtained using the traditional covariance estimate. This improvement shows the robustness of the denoised covariance estimate.

Sample Size Selection for Covariance Denoising

This example shows the effect of the sample size in `covarianceDenoising` to compute a denoised version of an initial covariance estimate. The `covarianceDenoising` function has two different syntaxes. This example focuses on the syntax with two inputs: a covariance estimate and the size of the sample that you use to compute the initial covariance estimate. This example demonstrates an empirical study of the effect of the sample size on the acuracy of the denoised estimate. The goal of this example is to provide insight for the choice of the sample size when this information is not available. This scenario can happen if the initial covariance estimate is stored but the information of the sample used to compute the estimate is not available.

Create Random Sample

Create a random sample using `mvnrnd`.

```
% Load data
load('port5.mat','mean_return','Correlation')

% Select the number of assets
numObservations = 300;
numAssets = 100;
mu = mean_return(1:numAssets);
C = Correlation(1:numAssets,1:numAssets);

% Generate a random sample
rng('default')
randSample = mvnrnd(mu,C,numObservations);
```

Use `corr` to compute the standard correlation estimate of the sample. This example works with the correlation matrix to remove the effect of the variance. This is useful when comparing the effect of the sample size in the denoised covariance estimate.

```
stdCorr = corr(randSample);
```

Due to numerical errors, `stdCorr` is not symmetric. Modify `stdCorr` to make it symmetric.

```
% Compute maximum absolute difference of matrix against its
% transpose
max(abs(stdCorr-stdCorr'),[],'all')
```

```
ans = 1.1102e-16
```

```
% Force correlation matrix to be symmetric
stdCorr = (stdCorr+stdCorr')/2;
```

Use `covarianceDenoising` to compute the denoised covariance estimate from `Sigma` using the correct sample size.

```
denoisedCorr = covarianceDenoising(stdCorr,numObservations);
denoisedCorr = (denoisedCorr+denoisedCorr')/2;
```

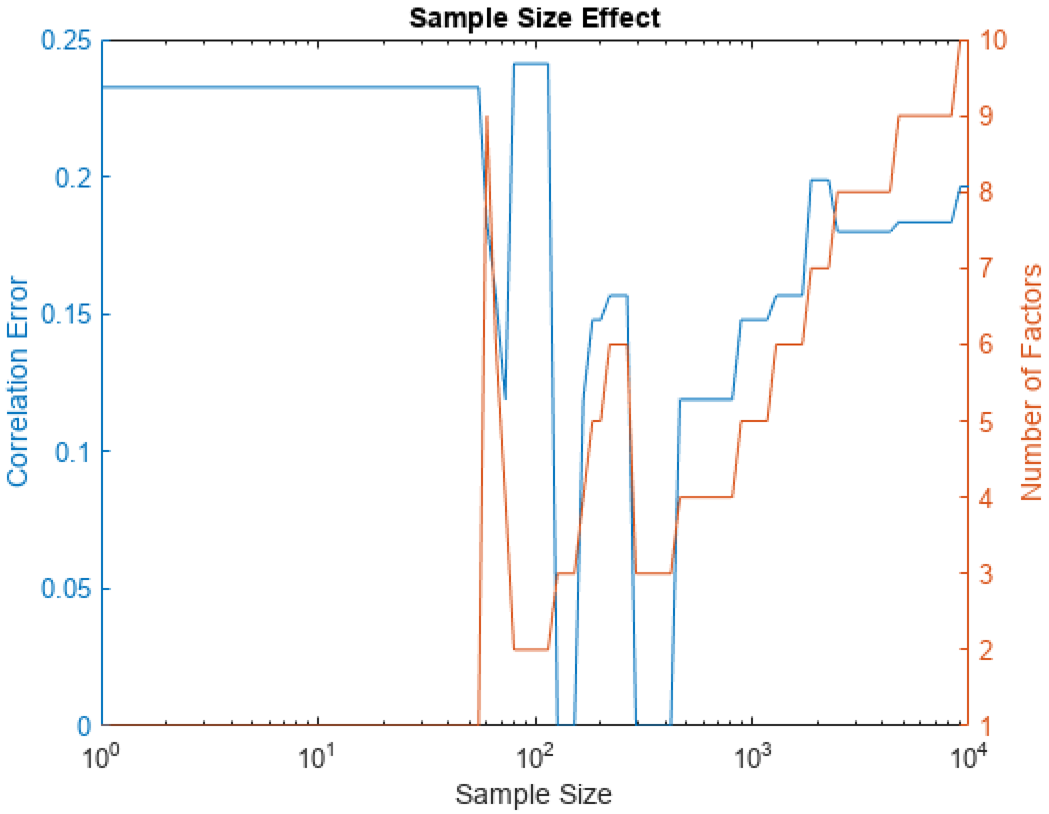
Compute Denoised Covariance for Different Sample Sizes

Compare the maximum correlation difference between the denoised covariance with the correct sample size and other sample sizes. Also, you compute the number of factors.

```
% Sample sizes to try
numSizes = 100;
sampleSizes = ceil(logspace(0,4,numSizes));

% Compute denoised correlation and store the maximum correlation
% mismatch
correlations = cell(numSizes);
numFactors = zeros(numSizes,1);
maxDiff = zeros(numSizes,1);
for i = 1:numSizes
    [correlations{i},numFactors(i)] = covarianceDenoising(stdCorr,...
        sampleSizes(i));
    maxDiff(i) = norm(correlations{i}(:)-denoisedCorr(:),inf);
end
```

Plot the maximum error in the correlations as well as the number of factors identified for each sample size.

```
figure;

% Plot correlation error
yyaxis left
semilogx(sampleSizes,maxDiff)
hold on
ylabel("Correlation Error");

% Plot number of factors
yyaxis right
semilogx(sampleSizes,numFactors)
ylabel("Number of Factors");

% Title, labels, and legends
title("Sample Size Effect")
xlabel("Sample Size");
hold off
```



Note that when the sample size is close to the number of assets, the behavior of both the correlation error and the number of factors is erratic. Here, when the sample size is between 55 and 292, the number of factors and the correlation error jump up and down. Before 55 "observations" the number of factors is 1 because the denoising method doesn't have enough information to trust that what is being observed is signal and not noise; only the largest eigenvalue is associated with signal. Usually the largest eigenvalue is linked to the market. Meanwhile, after 292 "observations" the number of eigenvalues not associated with noise monotonically increases. This behavior happens because denoising assumes that the sample is less noisy with more observations, so identifying signal from noise is easier.

If the sample size of the initial covariance estimate is not known, it is better to choose a sample size that is far from the number of assets. For cases with less trust in the initial covariance estimate, the sample size should be smaller, below 55 in this example. If you trust the initial covariance estimate, then the sample size should be larger, above 292 in this example. The degree of trust in the initial covariance matrix can be adjusted by increasing the sample size.

Backtest to Compare Investment Strategy When Using Covariance Denoising

This example compares a minimum variance investment strategy using the traditional covariance estimate with a minimum variance strategy using covariance denoising.

Load Data

```
% Read a table of daily adjusted close prices for 2006 DJIA stocks.
T = readtable('dowPortfolio.xlsx');

% Convert the table to a timetable.
pricesTT = table2timetable(T,'RowTimes','Dates');
numAssets = size(pricesTT.Variables, 2);
```

Use the first 42 days of the `dowPortfolio.xlsx` data set to initialize the backtest strategies. The backtest is then run over the remaining data.

```
warmupPeriod = 42;
```

Compute Initial Weights

Use the `traditionalStrat` and `denoisedStrat` functions in Local Functions on page 15-53 to compute the weights.

```
% Specify no current weights (100% cash position).
w0 = zeros(1,numAssets);

% Specify warm-up partition of data set timetable.
warmupTT = pricesTT(1:warmupPeriod,:);

% Compute the initial portfolio weights for each strategy.
traditional_initial = traditionalStrat(w0,warmupTT);
shrunk_initial = denoisedStrat(w0,warmupTT);
```

Create Backtest Strategies

Create `Traditional` and `Denoising` backtest strategy objects using `backtestStrategy`.

```
% Rebalance approximately every month.
rebalFreq = 21;

% Set the rolling lookback window to be at least 2 months and at
% most 6 months.
lookback  = [42 126];

% Use a fixed transaction cost (buy and sell costs are both 0.5%
% of amount traded).
```

```
transactionsFixed = 0.005;

% Specify the strategy objects.
strat1 = backtestStrategy('Traditional', @traditionalStrat, ...
    RebalanceFrequency=rebalFreq, ...
    LookbackWindow=lookback, ...
    TransactionCosts=transactionsFixed, ...
    InitialWeights=traditional_initial);

strat2 = backtestStrategy('Denoising', @denoisedStrat, ...
    RebalanceFrequency=rebalFreq, ...
    LookbackWindow=lookback, ...
    TransactionCosts=transactionsFixed, ...
    InitialWeights=shrunk_initial);

% Aggregate the two strategy objects into an array.
strategies = [strat1, strat2];
```

Create a `backtestEngine` object then use `runBacktest` to run the backtest.

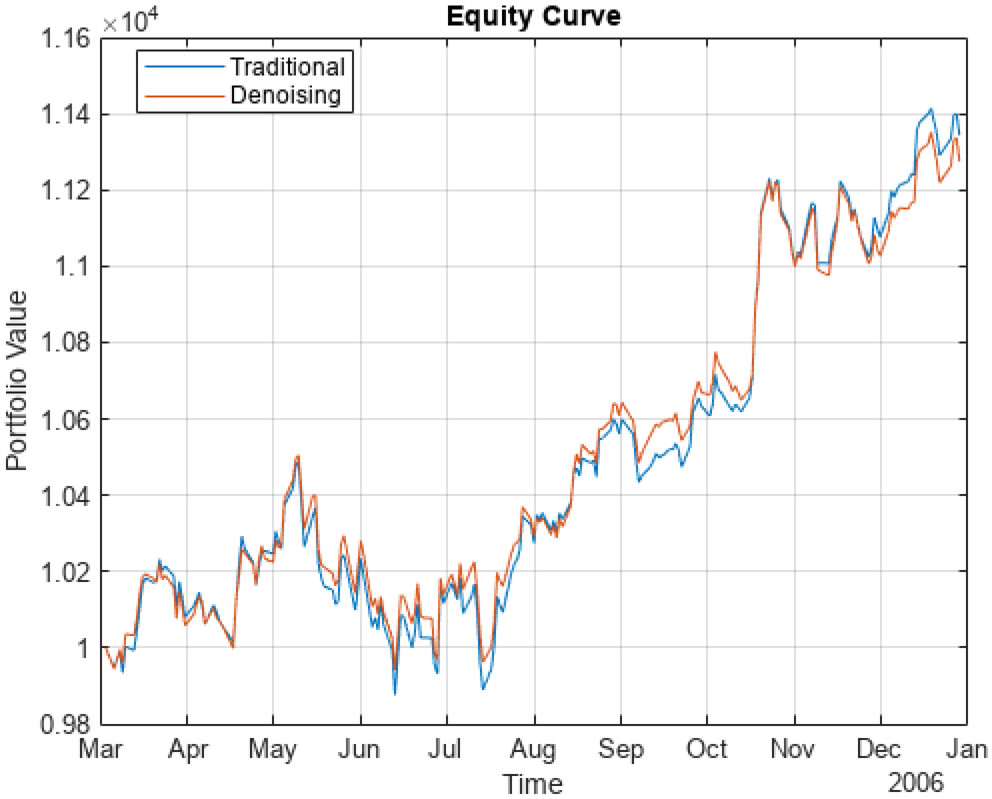
```
% Create the backtesting engine object.
backtester = backtestEngine(strategies);

% Run the backtest.
backtester = runBacktest(backtester,pricesTT,'Start',warmupPeriod);

% Generate summary table of the performance of the strategies.
summary(backtester)
```

```
ans=9×2 table
                       Traditional    Denoising 
                       ___________    __________

    TotalReturn            0.13431       0.12745
    SharpeRatio            0.10807       0.10719
    Volatility           0.0057472     0.0055101
    AverageTurnover      0.0098378     0.0074213
    MaxTurnover            0.36237        0.2791
    AverageReturn        0.0006196    0.00058922
    MaxDrawdown           0.058469      0.053624
    AverageBuyCost         0.51636       0.38982
    AverageSellCost        0.51636       0.38982
```

Compare Performance of Strategies

Use `equityCurve` to plot the equity curve to compare the performance of both strategies.

```
equityCurve(backtester)
```



The maximum and average turnover are decreased using covariance denoising. Also, the covariance denoising strategy results in a decrease of buy and sell costs. In this example, not only is the turnover decreased, but also the volatility and the maximum drawdown are decreased. Therefore, in this example the denoised covariance produces more robust weights.

Local Functions

```
function new_weights = traditionalStrat(~, pricesTT)
% Function for minimum variance portfolio using traditional covariance estimate.

% Compute the returns from the prices timetable.
assetReturns = tick2ret(pricesTT);
mu = mean(assetReturns.Variables);
Sigma = cov(assetReturns.Variables,"omitrows");

% Create the portfolio problem.
p = Portfolio(AssetMean=mu,AssetCovar=Sigma);
% Specify long-only fully invested contraints.
p = setDefaultConstraints(p);

% Compute the minimum variance portfolio.
new_weights = estimateFrontierLimits(p,'min');
end

function new_weights = denoisedStrat(~, pricesTT)
% Function for minimum variance portfolio using covariance denoising.
% Compute the returns from the prices timetable.
assetReturns = tick2ret(pricesTT);
mu = mean(assetReturns.Variables);
Sigma = covarianceDenoising(assetReturns.Variables);

% Create the portfolio problem.
p = Portfolio(AssetMean=mu,AssetCovar=Sigma);
% Specify long-only fully invested contraints.
p = setDefaultConstraints(p);

% Compute the minimum variance portfolio.
new_weights = estimateFrontierLimits(p,'min');
end
```

Input Arguments

AssetReturns — Asset returns

matrix | table | timetable

Asset returns, specified as a NumObservations-by-NumAssets matrix, table, or timetable.

Note All rows (observations) with one or more NaN values are removed before computing the covariance estimate.

Data Types: double | table | timetable

Sigma — Initial covariance matrix estimate

matrix

Covariance matrix estimate, specified as a NumAssets-by-NumAssets covariance estimate matrix.

Data Types: double

SampleSize — Sample size to compute initial covariance estimate

scalar numeric

Sample size to compute the initial covariance estimate Sigma, specified as a scalar numeric value.

Data Types: double

Output Arguments

SigmaHat — Estimated covariance matrix

matrix

Estimated covariance matrix, returned as a NumAssets-by-NumAssets matrix.

numSignalEig — Number of eigenvalues associated with signal

numeric

Number of eigenvalues associated with signal, returned as a scalar numeric.

More About

Covariance Denoising

The goal of covariance denoising is to reduce the noise and enhance the signal of the empirical covariance matrix.

Covariance denoising is a common method that is used to reduce the effect of noise in the covariance approximation. Covariance denoising reduces the ill-conditioning of the traditional covariance estimate by differentiating the eigenvalues associated with noise from the eigenvalues associated with signal. Random matrix theory is used in covariance denoising to pull only the eigenvalues that are associated with noise to a common mean.

An intuitive example is when the number of variables is larger than the number of observations. In this case, the result is a noninvertible covariance matrix. Furthermore, because covariance estimation is performed using observations from random data, the estimator contains a certain amount of noise. This noise usually results in ill-conditioned covariance estimates. Working with an ill-conditioned matrix magnifies the impact of estimation errors.

Unlike shrinkage methods, denoising does not require information from the full sample. In covariance denoising using `covarianceDenoising` you only need to know the number of scenarios that were used to estimate the covariance matrix.

Algorithms

The `covarianceDenoising` function shrinks only the part of the covariance that corresponds with noise as follows:

1. Compute the correlation matrix C associated with the traditional covariance estimate Σ.
2. Compute the eigendecomposition of $C = V\Lambda V^T$.
3. Estimate the empirical distribution of the eigenvalues using kernel density estimation with `fitdist(x,'Kernal')`. For more information, see `fitdist`.
4. Fit the Marchenko-Pastur distribution to the empirical distribution by minimizing the mean squared error (MSE) between the empirical probability density function (pdf) and the fitted Marchenko-Pastur pdf. This gives the theoretical bounds λ^+ and λ^- on the eigenvalues associated with noise.
5. Let $\bar{\lambda}$ be the average of the eigenvalues smaller than λ^+. Set all eigenvalues smaller than λ^+ to $\bar{\lambda}$. These are the eigenvalues associated with noise.
6. Compute the denoised version of the correlation matrix $\widehat{C} = V\bar{\Lambda}V^T$ and rescale $\widehat{C}$ so that the main diagonal only has ones. $\widehat{C}$ is a correlation matrix.
7. Compute the denoised covariance estimate $\widehat{\Sigma}$ from $\widehat{C}$.

Version History

Introduced in R2023a

References

[1] Lòpez de Prado, M. *Machine Learning for Asset Managers (Elements in Quantitative Finance).* Cambridge University Press, 2020.

See Also

`covarianceShrinkage` | `corrcoef` | `partialcorr` | `corrcov` | `robustcov` | `nearcorr`

Topics

"Compare Performance of Covariance Denoising with Factor Modeling Using Backtesting" on page 4-378
"Comparison of Methods for Covariance Estimation" on page 4-134

covarianceShrinkage

Estimate covariance matrix using shrinkage estimators

Syntax

```
SigmaHat = covarianceShrinkage(AssetReturns)
```

Description

`SigmaHat = covarianceShrinkage(AssetReturns)` returns a covariance estimate using linear shrinkage to reduce the mean squared error (MSE).

`covarianceShrinkage` computes an estimate of the covariance matrix from a sample of asset returns using the multiple of the identity shrinkage estimation method. For more information, see "Covariance Shrinkage" on page 15-62 and "covarianceShrinkage Algorithm" on page 15-63.

In addition, you can use `covarianceDenoising` to compute an estimate of covariance matrix using denoising. For information on which covariance estimation method to choosesee "Comparison of Methods for Covariance Estimation" on page 4-134.

Examples

Perform Covariance Linear Shrinkage

This example shows how to use `covarianceShrinkage` to compute covariance estimations that take into account the noise in the sample. In mean-variance portfolio optimization, a noisy estimate of the covariance estimate results in unstable solutions that cause high turnover and transaction costs. Ideally, to decrease the estimation error, it is desirable to increase the sample size. Yet, there are cases where this is not possible. In extreme cases in which the number of assets is larger than the number of observations, the traditional covariance matrix results in a singular matrix. Working with a nearly singular or an ill-conditioned covariance matrix magnifies the impact of estimation errors.

Compute a portfolio efficient frontier using different covariance estimates with the same sample data.

```
% Load portfolio data with 225 assets
load port5.mat
covariance = corr2cov(stdDev_return,Correlation);
% Generate a sample with 200 observations
rng('default')
nScen = 200;
retSeries = portsim(mean_return',covariance,nScen);
```

Compute the traditional and shrunk covariance estimates. Use `covarianceShrinkage` to reduce the effect of noise in the covariance approximation.

```
Sigma = cov(retSeries);
shrunkSigma = covarianceShrinkage(retSeries);
```

Compute the condition number of both covariance estimates. The shrunken covariance (`shrunkSigma`) has a lower condition number than the traditional covariance estimate `Sigma`.

```
conditionNum = [cond(Sigma); cond(shrunkSigma)];
condNumT = table(conditionNum,'RowNames',{'Sigma','SigmaHat'})
```

```
condNumT=2×1 table
                conditionNum
                ____________

    Sigma        2.5285e+18
    SigmaHat         5274.8
```

Use Portfolio to construct Portfolio objects that use the different AssetCovar values. Then use setDefaultConstraints to set the portfolio constraints with nonnegative weights that sum to 1 for the three portfolios: the true mean with the true covariance, the traditional covariance estimate, and the shrunk estimate.

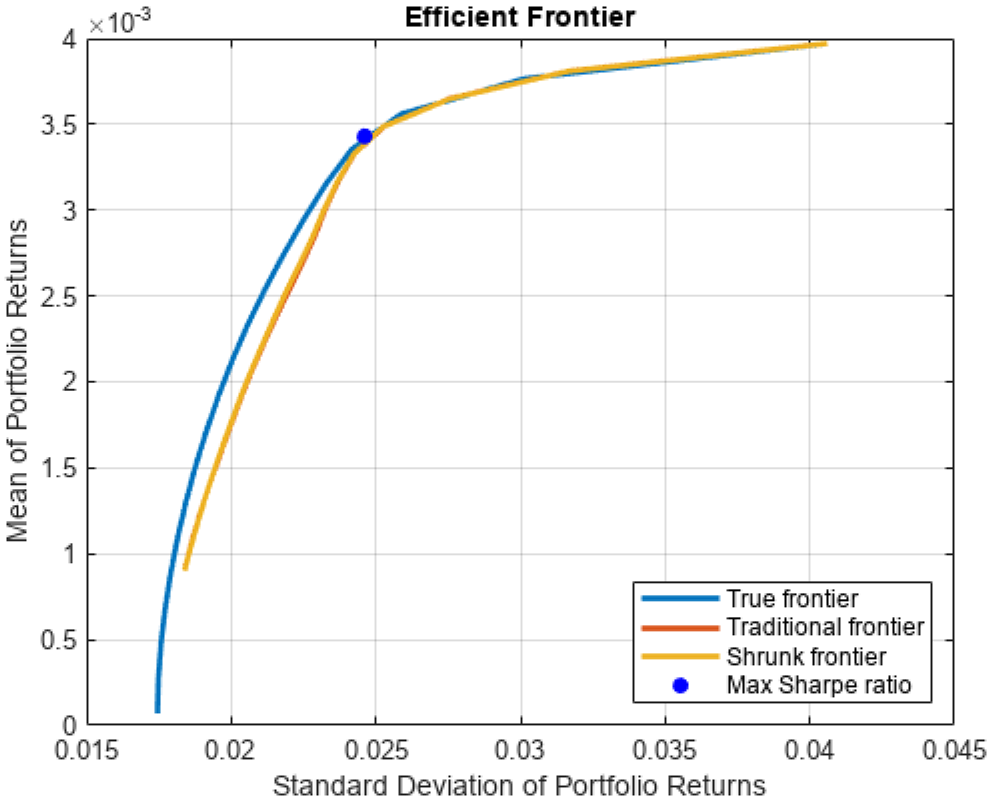
```
% Create a Portfolio object with the true parameters
p = Portfolio(AssetMean=mean_return,AssetCovar=covariance);
p = setDefaultConstraints(p);

% Create a Portfolio object with true mean and traditional covariance estimate
pTraditional = Portfolio(AssetMean=mean_return,AssetCovar=Sigma);
pTraditional = setDefaultConstraints(pTraditional);

% Create a Portfolio object with true mean and shrunk covariance
pShrunk = Portfolio(AssetMean=mean_return,AssetCovar=shrunkSigma);
pShrunk = setDefaultConstraints(pShrunk);
```

Use estimateFrontier to estimate the efficient frontier for each of the Portfolio objects.

```
% Number of portfolios on the efficient frontier
nPort = 20;

% True efficient portfolios
w = estimateFrontier(p,nPort);

% Traditional covariance efficient portfolios
wTraditional = estimateFrontier(pTraditional,nPort);

% Denoised covariance efficient portfolios
wShrunk = estimateFrontier(pShrunk,nPort);
```

Use plotFrontier to plot the frontier obtained from the different weights using the true parameter values.

```
figure
plotFrontier(p,w)
hold on
plotFrontier(p,wTraditional)
plotFrontier(p,wShrunk)
plotFrontier(p,estimateMaxSharpeRatio(p))
legend('True frontier','Traditional frontier','Shrunk frontier', ...,
'Max Sharpe ratio',Location='southeast');
hold off
```



In this example, the efficient frontiers obtained using the traditional covariance estimate and the shrunk estimate are close to each other. This means that both methods achieve similar risk and returns out-of-sample. Where the difference between these methods is more noticeable is for the portfolios to the left of the maximum Sharpe ratio portfolio. For those portfolios, the allocation computed using shrinkage has better returns out-of-sample.

Backtest to Compare Investment Strategy When Using Covariance Shrinkage

This example compares a minimum variance investment strategy using the traditional covariance estimate with a minimum variance strategy using covariance shrinkage.

Load the data.

```
% Read a table of daily adjusted close prices for 2006 DJIA stocks.
T = readtable('dowPortfolio.xlsx');

% Convert the table to a timetable.
pricesTT = table2timetable(T,'RowTimes','Dates');
numAssets = size(pricesTT.Variables, 2);
```

Use the first 42 days of the `dowPortfolio.xlsx` data set to initialize the backtest strategies. The backtest is then run over the remaining data.

```
warmupPeriod = 42;
```

Compute the initial weights. Use the `traditionalStrat` and `shrunkStrat` functions in Local Functions on page 15-61 to compute the weights.

```
% Specify no current weights (100% cash position).
w0 = zeros(1,numAssets);

% Specify warm-up partition of data set timetable.
warmupTT = pricesTT(1:warmupPeriod,:);

% Compute the initial portfolio weights for each strategy.
traditional_initial = traditionalStrat(w0,warmupTT);
shrunk_initial = shrunkStrat(w0,warmupTT);
```

Create traditional and shrinkage backtest strategy objects using `backtestStrategy`.

```
% Rebalance approximately every month.
rebalFreq = 21;

% Set the rolling lookback window to be at least 2 months and at
% most 6 months.
lookback  = [42 126];

% Use a fixed transaction cost (buy and sell costs are both 0.5%
% of amount traded).
transactionsFixed = 0.005;

% Specify the strategy objects.
strat1 = backtestStrategy('Traditional', @traditionalStrat, ...
    RebalanceFrequency=rebalFreq, ...
    LookbackWindow=lookback, ...
    TransactionCosts=transactionsFixed, ...
    InitialWeights=traditional_initial);

strat2 = backtestStrategy('Shrinkage', @shrunkStrat, ...
    RebalanceFrequency=rebalFreq, ...
    LookbackWindow=lookback, ...
    TransactionCosts=transactionsFixed, ...
    InitialWeights=shrunk_initial);

% Aggregate the two strategy objects into an array.
strategies = [strat1, strat2];
```

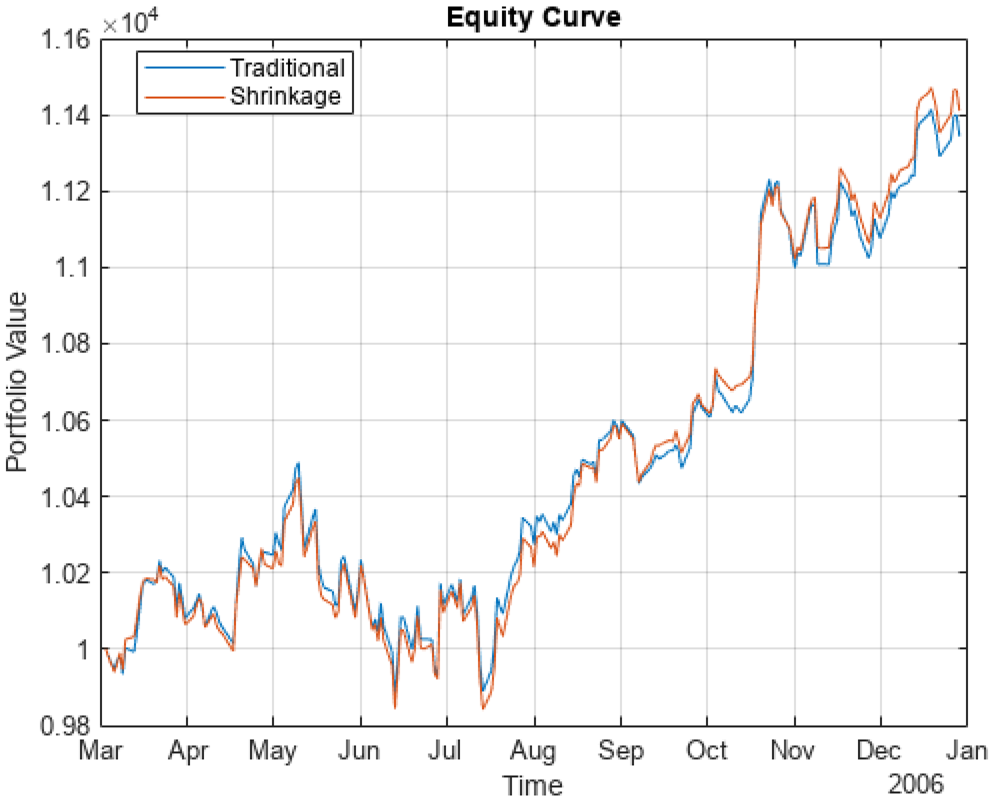
Create a `backtestEngine` object then use `runBacktest` to run the backtest.

```
% Create the backtesting engine object.
backtester = backtestEngine(strategies);

% Run the backtest.
backtester = runBacktest(backtester,pricesTT,'Start',warmupPeriod);

% Generate summary table of the performance of the strategies.
summary(backtester)
```

```
ans=9×2 table
                       Traditional    Shrinkage
                       ___________    _________

    TotalReturn          0.13431        0.14101
    SharpeRatio          0.10807        0.11565
    Volatility         0.0057472      0.0056077
    AverageTurnover    0.0098378      0.0077384
    MaxTurnover          0.36237        0.31835
    AverageReturn      0.0006196     0.00064699
    MaxDrawdown         0.058469       0.058133
    AverageBuyCost       0.51636        0.40533
    AverageSellCost      0.51636        0.40533
```

Use `equityCurve` to plot the equity curve to compare the performance of both strategies.

```
equityCurve(backtester)
```



The maximum and average turnover are decreased using covariance shrinkage. Also, the covariance shrinkage strategy results in a decrease of buy and sell costs. In this example, not only is the turnover decreased, but also the volatility and the maximum drawdown are decreased. Therefore, in this example the shrunk covariance produces more robust weights.

Local Functions

```
function new_weights = traditionalStrat(~, pricesTT)
% Function for minimum variance portfolio using traditional covariance estimate.

% Compute the returns from the prices timetable.
assetReturns = tick2ret(pricesTT);
mu = mean(assetReturns.Variables);
Sigma = cov(assetReturns.Variables,"omitrows");
```

```
% Create the portfolio problem.
p = Portfolio(AssetMean=mu,AssetCovar=Sigma);
% Specify long-only fully invested contraints.
p = setDefaultConstraints(p);

% Compute the minimum variance portfolio.
new_weights = estimateFrontierLimits(p,'min');
end

function new_weights = shrunkStrat(~, pricesTT)
% Function for minimum variance portfolio using covariance shrinkage.

% Compute the returns from the prices timetable.
assetReturns = tick2ret(pricesTT);
mu = mean(assetReturns.Variables);
Sigma = covarianceShrinkage(assetReturns.Variables);

% Create the portfolio problem.
p = Portfolio(AssetMean=mu,AssetCovar=Sigma);
% Specify long-only fully invested contraints.
p = setDefaultConstraints(p);

% Compute the minimum variance portfolio.
new_weights = estimateFrontierLimits(p,'min');
end
```

Input Arguments

AssetReturns — Asset returns

matrix | table | timetable

Asset returns, specified as a NumObservations-by-NumAssets matrix, table, or timetable.

Note All NumObservations with one or more NaN values are removed before computing the covariance estimate.

Data Types: double | table | timetable

Output Arguments

SigmaHat — Covariance estimate

matrix

Covariance estimate, returned as a NumAssets-by-NumAssets matrix.

More About

Covariance Shrinkage

Shrinkage estimators for covariance shrinkage are used to reduce the effect of noise in the covariance approximation.

The goal of covariance shrinkage is to pull all eigenvalues of the traditional covariance matrix towards a target.

Algorithms

The `covarianceShrikage` function applies a linear shrinkage method that shrinks the traditional covariance estimate to a multiple of the identity matrix.

$$\widehat{\Sigma} = (1 - \alpha)\Sigma + \alpha(\tau I)$$

Here, Σ is the standard covariance estimate, τ is the average sample variance, and $\alpha \in [0, 1]$ is the intensity parameter computed using

$$\alpha = \frac{\frac{1}{N}\sum_{i = 1}^{N} \text{trace}\left(\left[z_i z_i^T - \Sigma\right]^2\right)}{\text{trace}([\Sigma\text{-}\tau]^2)}$$

where z_i is the i th row of the centered sample matrix Z and N is the sample size.

Version History

Introduced in R2023a

References

[1] Ledoit, O. and Wolff, M. "A Well-Conditioned Estimator for Large-Dimensional Covariance Matrices." *Journal of Multivariate Analysis.* vol. 88, no. 2, 365–411, 2004.

See Also

`covarianceDenoising` | `corrcoef` | `partialcorr` | `corrcov` | `robustcov` | `nearcorr`

Topics

"Compare Performance of Covariance Denoising with Factor Modeling Using Backtesting" on page 4-378
"Comparison of Methods for Covariance Estimation" on page 4-134

bm

Brownian motion (BM) models

Description

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.

Use bm 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.

Creation

Syntax

```
BM = bm(Mu,Sigma)
BM = bm( ___ ,Name,Value)
```

Description

BM = bm(Mu,Sigma) creates a default BM object.

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.

`BM = bm(___,Name,Value)` creates 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`

The `BM` object has the following “Properties” on page 15-66:

- `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

Input Arguments

`Mu` — `Mu` represents the parameter μ

array or deterministic function of time | 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 | deterministic function of time and state

`Sigma` represents the parameter *V*, specified as an array or 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

Properties

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 numeric.

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, 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.

Data Types: double

Correlation — Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes)
NBrowns-by-NBrowns identity matrix representing independent Gaussian processes (default) | positive semidefinite matrix | deterministic function

Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes), specified 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. If Correlation is not a symmetric positive semidefinite matrix, use nearcorr to create a positive semidefinite matrix for a correlation matrix.

A Correlation matrix represents a static condition.

As a deterministic function of time, Correlation allows you to specify a dynamic correlation structure.

Data Types: double

Simulation — User-defined simulation function or SDE simulation method
simulation by Euler approximation (simByEuler) (default) | function | SDE simulation method

User-defined simulation function or SDE simulation method, specified as a function or SDE simulation method.

Data Types: `function_handle`

`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)

This property is read-only.

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 `drift` 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 displayed parameters for a `drift` object 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)`

Data Types: `object`

`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)

This property is read-only.

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 `diffusion`:

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

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 displayed parameters for a `diffusion` object 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)`

Data Types: `object`

Object Functions

interpolate	Brownian interpolation of stochastic differential equations (SDEs) for SDE, BM, GBM, CEV, CIR, HWV, Heston, SDEDDO, SDELD, or SDEMRD models
simulate	Simulate multivariate stochastic differential equations (SDEs) for SDE, BM, GBM, CEV, CIR, HWV, Heston, SDEDDO, SDELD, SDEMRD, Merton, or Bates models
simByEuler	Euler simulation of stochastic differential equations (SDEs) for SDE, BM, GBM, CEV, CIR, HWV, Heston, SDEDDO, SDELD, or SDEMRD models
simByMilstein	Simulate BM, GBM, CEV, HWV, SDEDDO, SDELD, SDEMRD sample paths by Milstein approximation

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:

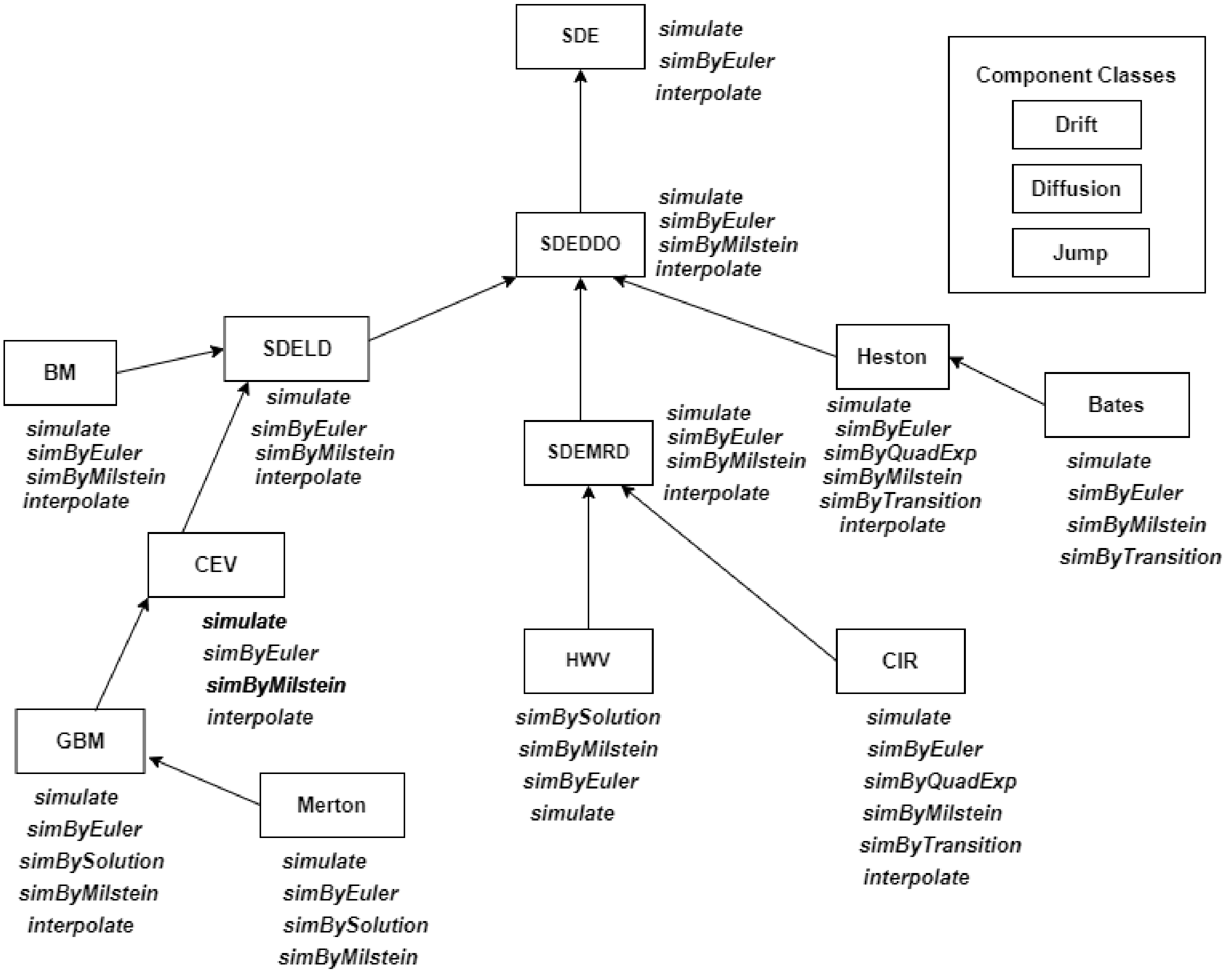
- The expected drift, or trend, rate Mu is a column vector.
- The volatility rate, Sigma, is a matrix.
- No end-of-period adjustments and/or processes are made.
- If specified, the random noise process Z is a three-dimensional array.
- If Z is unspecified, the assumed Gaussian correlation structure is a double matrix.

More About

Instance Hierarchy

There are inheritance relationships among the SDE classes.

The following figure illustrates the inheritance relationships.



For more information, see “SDE Class Hierarchy” on page 14-5.

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.

Version History

Introduced in R2008a

References

[1] Aït-Sahalia, Yacine. “Testing Continuous-Time Models of the Spot Interest Rate.” *Review of Financial Studies*, vol. 9, no. 2, Apr. 1996, pp. 385–426.

[2] Aït-Sahalia, Yacine. “Transition Densities for Interest Rate and Other Nonlinear Diffusions.” *The Journal of Finance*, vol. 54, no. 4, Aug. 1999, pp. 1361–95.

[3] Glasserman, Paul. *Monte Carlo Methods in Financial Engineering*. Springer, 2004.

[4] Hull, John. *Options, Futures and Other Derivatives*. 7th ed, Prentice Hall, 2009.

[5] Johnson, Norman Lloyd, et al. *Continuous Univariate Distributions*. 2nd ed, Wiley, 1994.

[6] Shreve, Steven E. *Stochastic Calculus for Finance*. Springer, 2004.

See Also

`drift` | `diffusion` | `sdeld` | `simulate` | `interpolate` | `simByEuler` | `nearcorr`

Topics

“Simulating Equity Prices” on page 14-28
“Simulating Interest Rates” on page 14-48
“Stratified Sampling” on page 14-57
“Price American Basket Options Using Standard Monte Carlo and Quasi-Monte Carlo Simulation” on page 14-70
“Base SDE Models” on page 14-14
“Drift and Diffusion Models” on page 14-16
“Linear Drift Models” on page 14-19
“Parametric Models” on page 14-21
“SDEs” on page 14-2
“SDE Models” on page 14-7
“SDE Class Hierarchy” on page 14-5
“Quasi-Monte Carlo Simulation” on page 14-62
“Performance Considerations” on page 14-64

cev

Constant Elasticity of Variance (CEV) model

Description

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 CEV 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.

Creation

Syntax

```
CEV = cev(Return,Alpha,Sigma)
CEV = cev( ___ ,Name,Value)
```

Description

`CEV = cev(Return,Alpha,Sigma)` creates a default `CEV` object.

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.

`CEV = cev( ___ ,Name,Value)` creates 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`

The `CEV` object has the following "Properties" on page 15-74:

- `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

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`

Note Although `cev` does not enforce restrictions on the signs of these input arguments, each argument is specified as a positive value.

Properties

`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

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, `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.

Data Types: `double`

`Correlation` — Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes)

`NBrowns`-by-`NBrowns` identity matrix representing independent Gaussian processes (default) | positive semidefinite matrix | deterministic function

Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes), specified 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. If `Correlation` is not a symmetric positive semidefinite matrix, use `nearcorr` to create a positive semidefinite matrix for a correlation matrix.

A `Correlation` matrix represents a static condition.

As a deterministic function of time, `Correlation` allows you to specify a dynamic correlation structure.

Data Types: `double`

`Simulation` — User-defined simulation function or SDE simulation method

simulation by Euler approximation (`simByEuler`) (default) | function | SDE simulation method

User-defined simulation function or SDE simulation method, specified as a function or SDE simulation method.

Data Types: `function_handle`

`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)

This property is read-only.

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 `drift` 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 displayed parameters for a `drift` object 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)`

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)

This property is read-only.

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 diffusion:

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

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 displayed parameters for a diffusion object 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)`

Data Types: `struct` | `double`

Object Functions

interpolate — Brownian interpolation of stochastic differential equations (SDEs) for SDE, BM, GBM, CEV, CIR, HWV, Heston, SDEDDO, SDELD, or SDEMRD models

simulate — Simulate multivariate stochastic differential equations (SDEs) for SDE, BM, GBM, CEV, CIR, HWV, Heston, SDEDDO, SDELD, SDEMRD, Merton, or Bates models

simByEuler — Euler simulation of stochastic differential equations (SDEs) for SDE, BM, GBM, CEV, CIR, HWV, Heston, SDEDDO, SDELD, or SDEMRD models

simByMilstein — Simulate BM, GBM, CEV, HWV, SDEDDO, SDELD, SDEMRD sample paths by Milstein approximation

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
```

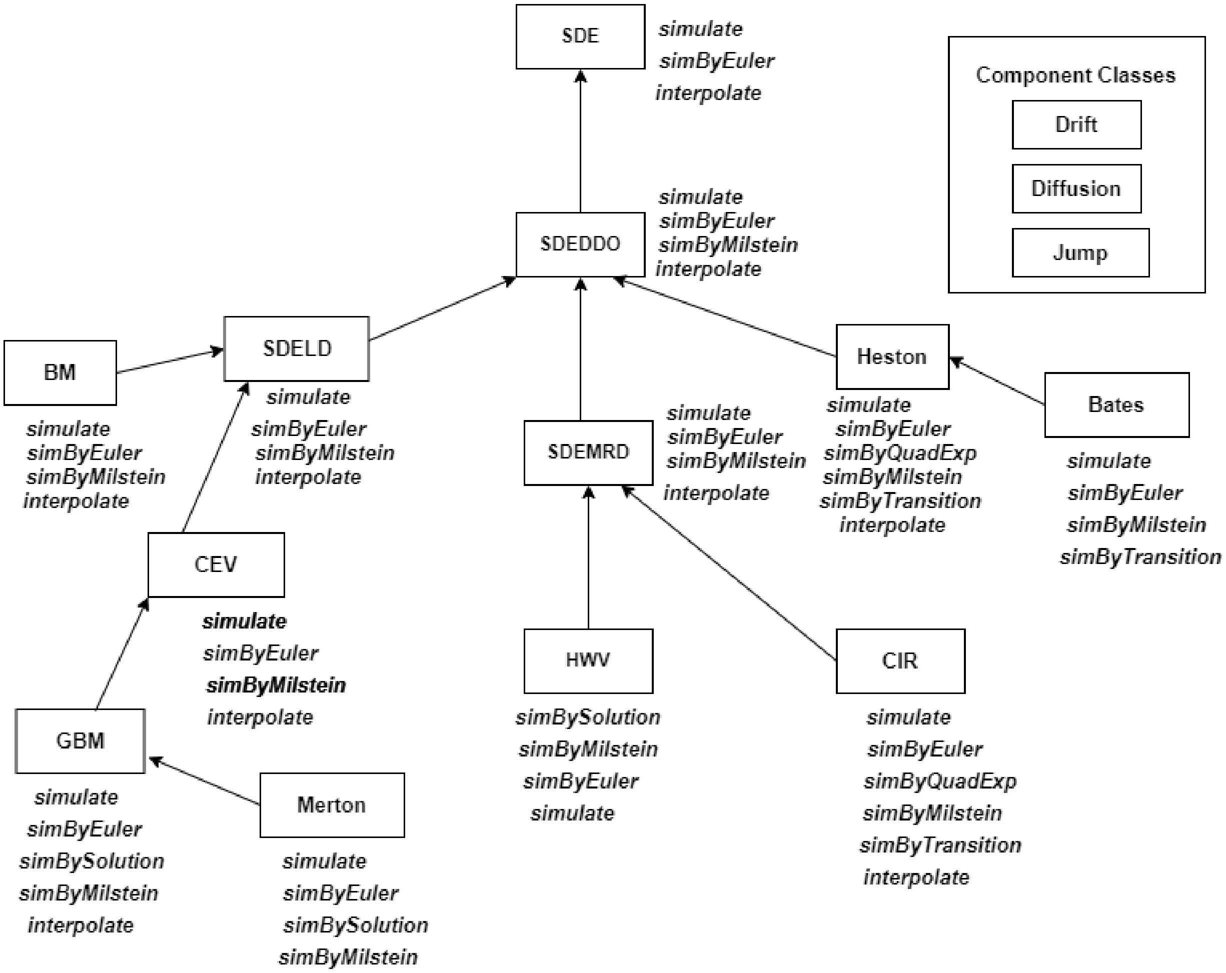
`cev` objects display the parameter `B` as the more familiar `Return`

More About

Instance Hierarchy

There are inheritance relationships among the SDE classes.

The following figure illustrates the inheritance relationships.



For more information, see “SDE Class Hierarchy” on page 14-5.

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.

Version History

Introduced in R2008a

References

[1] Aït-Sahalia, Yacine. “Testing Continuous-Time Models of the Spot Interest Rate.” *Review of Financial Studies*, vol. 9, no. 2, Apr. 1996, pp. 385–426.

[2] Aït-Sahalia, Yacine. “Transition Densities for Interest Rate and Other Nonlinear Diffusions.” *The Journal of Finance*, vol. 54, no. 4, Aug. 1999, pp. 1361–95.

[3] Glasserman, Paul. *Monte Carlo Methods in Financial Engineering*. Springer, 2004.

[4] Hull, John. *Options, Futures and Other Derivatives*. 7th ed, Prentice Hall, 2009.

[5] Johnson, Norman Lloyd, et al. *Continuous Univariate Distributions*. 2nd ed, Wiley, 1994.

[6] Shreve, Steven E. *Stochastic Calculus for Finance*. Springer, 2004.

See Also

`drift` | `diffusion` | `sdeld` | `simulate` | `interpolate` | `simByEuler` | `nearcorr`

Topics

“Creating Constant Elasticity of Variance (CEV) Models” on page 14-22
Implementing Multidimensional Equity Market Models, Implementation 3: Using SDELD, CEV, and GBM Objects on page 14-30
“Simulating Equity Prices” on page 14-28
“Simulating Interest Rates” on page 14-48
“Stratified Sampling” on page 14-57
“Price American Basket Options Using Standard Monte Carlo and Quasi-Monte Carlo Simulation” on page 14-70
“Base SDE Models” on page 14-14
“Drift and Diffusion Models” on page 14-16
“Linear Drift Models” on page 14-19
“Parametric Models” on page 14-21
“SDEs” on page 14-2
“SDE Models” on page 14-7
“SDE Class Hierarchy” on page 14-5
“Quasi-Monte Carlo Simulation” on page 14-62
“Performance Considerations” on page 14-64

cir

Cox-Ingersoll-Ross (`CIR`) mean-reverting square root diffusion model

Description

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.

You can simulate any vector-valued CIR process 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.

Creation

Syntax

```
CIR = cir(Speed,Level,Sigma)
CIR = cir( ___ ,Name,Value)
```

Description

`CIR = cir(Speed,Level,Sigma)` creates a default `CIR` object.

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.

`CIR = cir( ___ ,Name,Value)` creates 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`

The `CIR` object has the following “Properties” on page 15-82:

- `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

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`

Note Although `cir` does not enforce restrictions on the signs of these input arguments, each argument is specified as a positive value.

Properties

`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

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, 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.

Data Types: double

Correlation — Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes)
NBrowns-by-NBrowns identity matrix representing independent Gaussian processes (default) | positive semidefinite matrix | deterministic function

Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes), specified 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. If Correlation is not a symmetric positive semidefinite matrix, use nearcorr to create a positive semidefinite matrix for a correlation matrix.

A Correlation matrix represents a static condition.

As a deterministic function of time, Correlation allows you to specify a dynamic correlation structure.

Data Types: double

Simulation — User-defined simulation function or SDE simulation method
simulation by Euler approximation (simByEuler) (default) | function | SDE simulation method

User-defined simulation function or SDE simulation method, specified as a function or SDE simulation method.

Data Types: function_handle

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)

This property is read-only.

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 drift 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 displayed parameters for a drift object 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)`

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)

This property is read-only.

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 `diffusion`:

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

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 displayed parameters for a `diffusion` object 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)`

Data Types: `struct` | `double`

Object Functions

interpolate	Brownian interpolation of stochastic differential equations (SDEs) for SDE, BM, GBM, CEV, CIR, HWV, Heston, SDEDDO, SDELD, or SDEMRD models
simulate	Simulate multivariate stochastic differential equations (SDEs) for SDE, BM, GBM, CEV, CIR, HWV, Heston, SDEDDO, SDELD, SDEMRD, Merton, or Bates models
simByEuler	Euler simulation of stochastic differential equations (SDEs) for SDE, BM, GBM, CEV, CIR, HWV, Heston, SDEDDO, SDELD, or SDEMRD models
simByTransition	Simulate CIR sample paths with transition density
simByQuadExp	Simulate Bates, Heston, and CIR sample paths by quadratic-exponential discretization scheme
simByMilstein	Simulate CIR sample paths by Milstein approximation

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
```

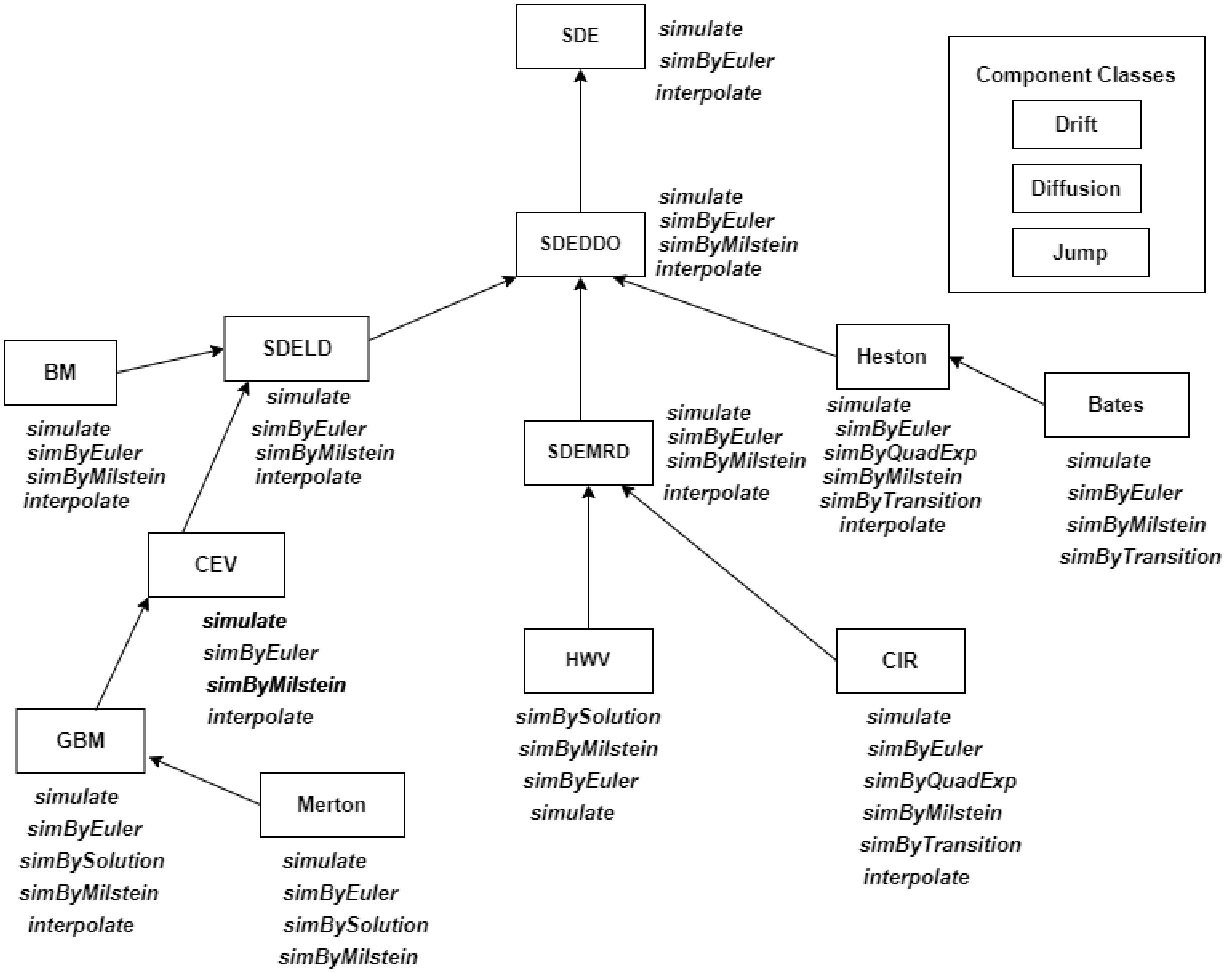
Although the last two objects are of different classes, they represent the same mathematical model. They differ in that you create the `cir` object by specifying only three input arguments. This distinction is reinforced by the fact that the `Alpha` parameter does not display – it is defined to be `1/2`.

More About

Instance Hierarchy

There are inheritance relationships among the SDE classes.

The following figure illustrates the inheritance relationships.



For more information, see “SDE Class Hierarchy” on page 14-5.

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.

Version History

Introduced in R2008a

R2023a: Added `simByMilstein` method

Behavior changed in R2023a

Use the `simByMilstein` method to approximate a numerical solution of a stochastic differential equation.

References

[1] Aït-Sahalia, Yacine. “Testing Continuous-Time Models of the Spot Interest Rate.” *Review of Financial Studies*, vol. 9, no. 2, Apr. 1996, pp. 385–426.

[2] Aït-Sahalia, Yacine. “Transition Densities for Interest Rate and Other Nonlinear Diffusions.” *The Journal of Finance*, vol. 54, no. 4, Aug. 1999, pp. 1361–95.

[3] Glasserman, Paul. *Monte Carlo Methods in Financial Engineering*. Springer, 2004.

[4] Hull, John. *Options, Futures and Other Derivatives*. 7th ed, Prentice Hall, 2009.

[5] Johnson, Norman Lloyd, et al. *Continuous Univariate Distributions*. 2nd ed, Wiley, 1994.

[6] Shreve, Steven E. *Stochastic Calculus for Finance*. Springer, 2004.

See Also

`drift` | `diffusion` | `sdeddo` | `simulate` | `interpolate` | `simByEuler` | `simByTransition` | `nearcorr`

Topics

“Creating Cox-Ingersoll-Ross (CIR) Square Root Diffusion Models” on page 14-24
“Simulating Equity Prices” on page 14-28
“Simulating Interest Rates” on page 14-48
“Stratified Sampling” on page 14-57

“Price American Basket Options Using Standard Monte Carlo and Quasi-Monte Carlo Simulation” on page 14-70
“Base SDE Models” on page 14-14
“Drift and Diffusion Models” on page 14-16
“Linear Drift Models” on page 14-19
“Parametric Models” on page 14-21
“SDEs” on page 14-2
“SDE Models” on page 14-7
“SDE Class Hierarchy” on page 14-5
“Quasi-Monte Carlo Simulation” on page 14-62
“Performance Considerations” on page 14-64

diffusion

Diffusion-rate model component

Description

The `diffusion` object 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:

$$G(t, X_t) = D(t, X_t^{\alpha(t)})V(t) \qquad (15\text{-}1)$$

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 α.
- 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.

Creation

Syntax

`DiffusionRate = diffusion(Alpha,Sigma)`

Description

`DiffusionRate = diffusion(Alpha,Sigma)` creates default `DiffusionRate` model component.

Specify required input parameters `A` and `B` 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 `diffusion` object that you create encapsulates the composite drift-rate specification and returns 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`.

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` two-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 `diffusion` 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)

This property is read-only.

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 $G(t,X_t)$.

Data Types: `struct` | `double`

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
```

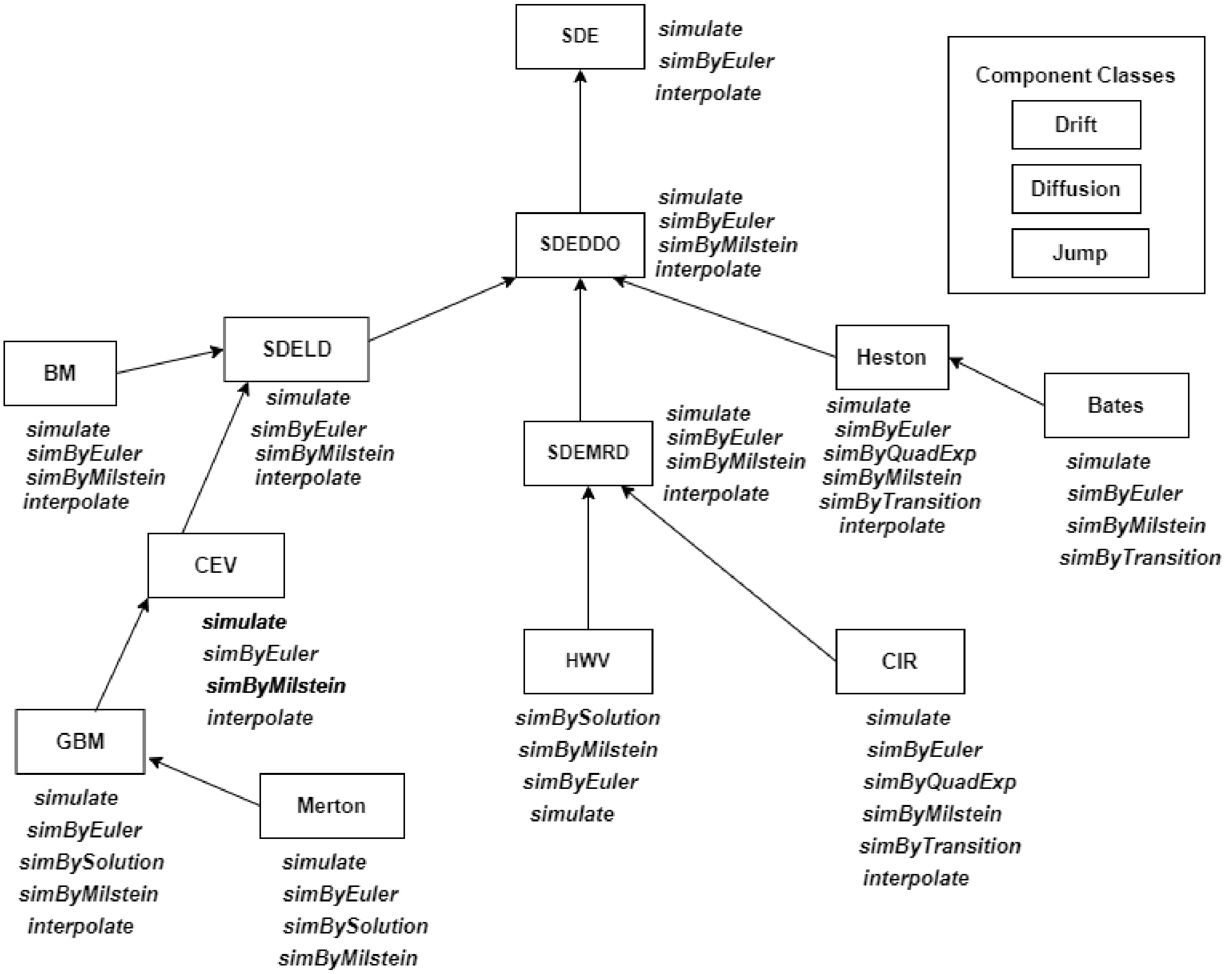
The `diffusion` object displays like a MATLAB® structure and contains supplemental information, namely, the object's class and a brief description. However, in contrast to the SDE representation, a summary of the dimensionality of the model does not appear, because the `diffusion` class creates a model component rather than a model. `G` does not contain enough information to characterize the dimensionality of a problem.

More About

Instance Hierarchy

There are inheritance relationships among the SDE classes.

The following figure illustrates the inheritance relationships.



For more information, see “SDE Class Hierarchy” on page 14-5.

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.

Version History

Introduced in R2008a

References

[1] Aït-Sahalia, Yacine. “Testing Continuous-Time Models of the Spot Interest Rate.” *Review of Financial Studies*, vol. 9, no. 2, Apr. 1996, pp. 385–426.

[2] Aït-Sahalia, Yacine. “Transition Densities for Interest Rate and Other Nonlinear Diffusions.” *The Journal of Finance*, vol. 54, no. 4, Aug. 1999, pp. 1361–95.

[3] Glasserman, Paul. *Monte Carlo Methods in Financial Engineering*. Springer, 2004.

[4] Hull, John. *Options, Futures and Other Derivatives*. 7th ed, Prentice Hall, 2009.

[5] Johnson, Norman Lloyd, et al. *Continuous Univariate Distributions*. 2nd ed, Wiley, 1994.

[6] Shreve, Steven E. *Stochastic Calculus for Finance*. Springer, 2004.

See Also

`drift` | `sdeddo`

Topics

“Simulating Equity Prices” on page 14-28
“Simulating Interest Rates” on page 14-48
“Stratified Sampling” on page 14-57
“Price American Basket Options Using Standard Monte Carlo and Quasi-Monte Carlo Simulation” on page 14-70
“Base SDE Models” on page 14-14
“Drift and Diffusion Models” on page 14-16
“Linear Drift Models” on page 14-19
“Parametric Models” on page 14-21

"SDEs" on page 14-2
"SDE Models" on page 14-7
"SDE Class Hierarchy" on page 14-5
"Performance Considerations" on page 14-64

drift

Drift-rate model component

Description

The `drift` object 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.

Creation

Syntax

```
DriftRate = drift(A,B)
```

Description

`DriftRate = drift(A,B)` creates a default `DriftRate` model component.

Specify required input parameters `A` and `B` 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 `drift` object that you create encapsulates the composite drift-rate specification and returns 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`.

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` two-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 $F(t,X_t)$

This property is read-only.

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:

- A: The intercept term, $A(t,X_t)$, of $F(t,X_t)$
- B: The first order term, $B(t,X_t)$, of $F(t,X_t)$

Data Types: struct | double

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
```

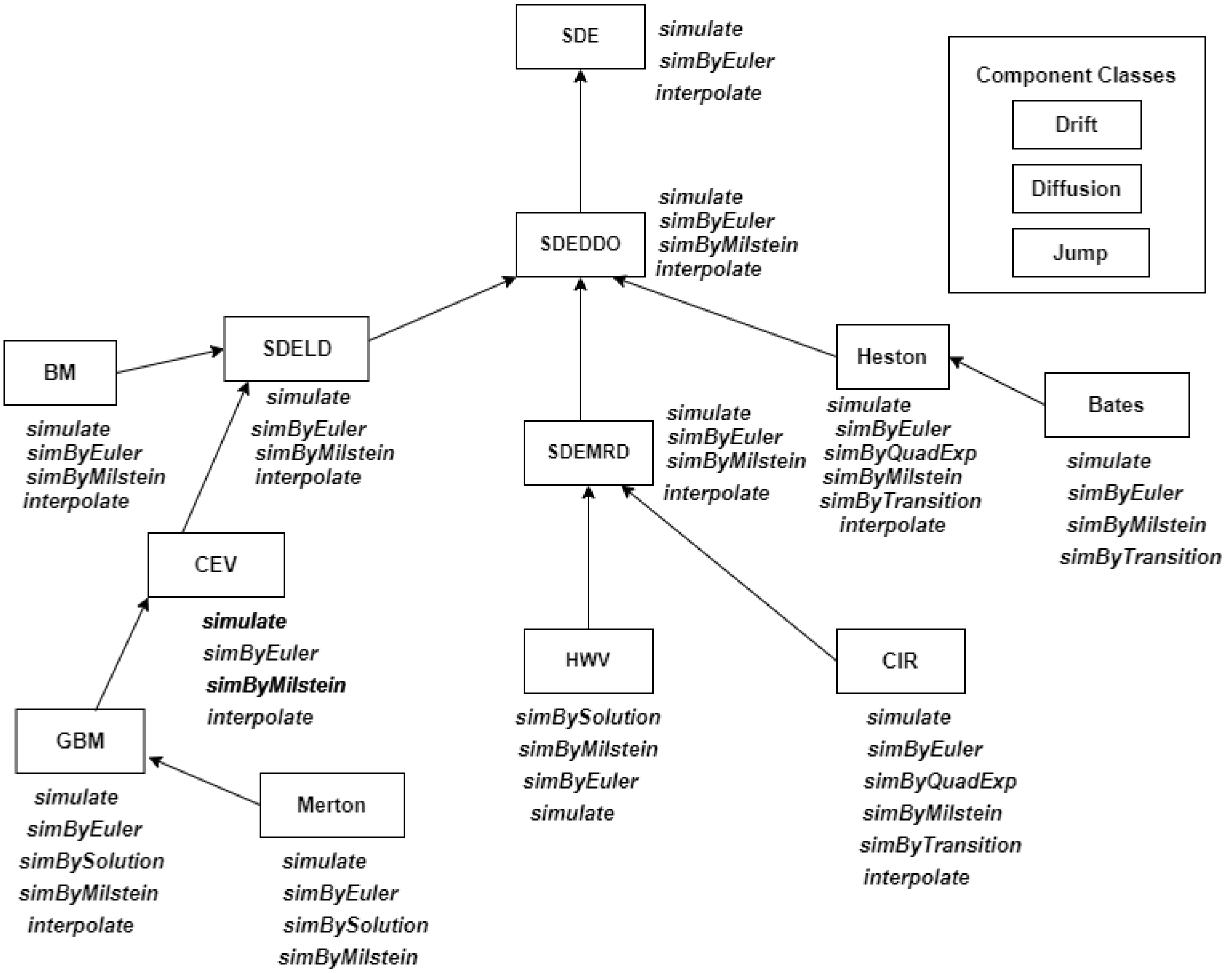
The drift object displays like a MATLAB® structure and contains supplemental information, namely, the object's class and a brief description. However, in contrast to the SDE representation, a summary of the dimensionality of the model does not appear, because the drift class creates a model component rather than a model. F does not contain enough information to characterize the dimensionality of a problem.

More About

Instance Hierarchy

There are inheritance relationships among the SDE classes.

The following figure illustrates the inheritance relationships.



For more information, see “SDE Class Hierarchy” on page 14-5.

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.

Version History

Introduced in R2008a

References

[1] Aït-Sahalia, Yacine. “Testing Continuous-Time Models of the Spot Interest Rate.” *Review of Financial Studies*, vol. 9, no. 2, Apr. 1996, pp. 385–426.

[2] Aït-Sahalia, Yacine. “Transition Densities for Interest Rate and Other Nonlinear Diffusions.” *The Journal of Finance*, vol. 54, no. 4, Aug. 1999, pp. 1361–95.

[3] Glasserman, Paul. *Monte Carlo Methods in Financial Engineering*. Springer, 2004.

[4] Hull, John. *Options, Futures and Other Derivatives*. 7th ed, Prentice Hall, 2009.

[5] Johnson, Norman Lloyd, et al. *Continuous Univariate Distributions*. 2nd ed, Wiley, 1994.

[6] Shreve, Steven E. *Stochastic Calculus for Finance*. Springer, 2004.

See Also

`diffusion` | `sdeddo`

Topics

“Simulating Equity Prices” on page 14-28
“Simulating Interest Rates” on page 14-48
“Stratified Sampling” on page 14-57
“Price American Basket Options Using Standard Monte Carlo and Quasi-Monte Carlo Simulation” on page 14-70
“Base SDE Models” on page 14-14
“Drift and Diffusion Models” on page 14-16
“Linear Drift Models” on page 14-19
“Parametric Models” on page 14-21
“SDEs” on page 14-2
“SDE Models” on page 14-7
“SDE Class Hierarchy” on page 14-5
“Performance Considerations” on page 14-64

gbm

Geometric Brownian motion (GBM) model

Description

Creates and displays geometric Brownian motion models, which derive from the cev (constant elasticity of variance) class.

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.

Creation

Syntax

```
GBM = gbm(Return,Sigma)
GBM = gbm( ___ ,Name,Value)
```

Description

GBM = gbm(Return,Sigma) creates a default GBM object.

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.

`GBM = gbm( ___ ,Name,Value)` creates 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`

The `GBM` object has the following “Properties” on page 15-102:

- `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

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 object enforces no restrictions on the sign of Sigma volatilities, they are specified as positive values.

Data Types: double | function_handle

Properties

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

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 object applies the same initial value to all state variables on all trials.

If StartState is a column vector, the gbm object applies a unique initial value to each state variable on all trials.

If StartState is a matrix, the gbm object applies a unique initial value to each state variable on each trial.

Data Types: double

Correlation — Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes)

NBrowns-by-NBrowns identity matrix representing independent Gaussian processes (default) | positive semidefinite matrix | deterministic function

Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes), specified 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. If Correlation is not a symmetric positive semidefinite matrix, use nearcorr to create a positive semidefinite matrix for a correlation matrix.

A Correlation matrix represents a static condition.

As a deterministic function of time, Correlation allows you to specify a dynamic correlation structure.

Data Types: double

`Simulation` — User-defined simulation function or SDE simulation method

simulation by Euler approximation (`simByEuler`) (default) | function | SDE simulation method

User-defined simulation function or SDE simulation method, specified as a function or SDE simulation method.

Data Types: `function_handle`

`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)

This property is read-only.

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 `drift`) 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 displayed parameters for a `drift` object 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)`

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 `diffusion`):

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

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)`

Data Types: `struct` | `double`

Object Functions

interpolate	Brownian interpolation of stochastic differential equations (SDEs) for SDE, BM, GBM, CEV, CIR, HWV, Heston, SDEDDO, SDELD, or SDEMRD models
simulate	Simulate multivariate stochastic differential equations (SDEs) for SDE, BM, GBM, CEV, CIR, HWV, Heston, SDEDDO, SDELD, SDEMRD, Merton, or Bates models
simByEuler	Euler simulation of stochastic differential equations (SDEs) for SDE, BM, GBM, CEV, CIR, HWV, Heston, SDEDDO, SDELD, or SDEMRD models
simBySolution	Simulate approximate solution of diagonal-drift GBM processes
simByMilstein	Simulate BM, GBM, CEV, HWV, SDEDDO, SDELD, SDEMRD sample paths by Milstein approximation

Examples

Create a gbm Object

Create a univariate gbm object to represent the model: $dX_t = 0.25X_tdt + 0.3X_tdW_t$.

```
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
```

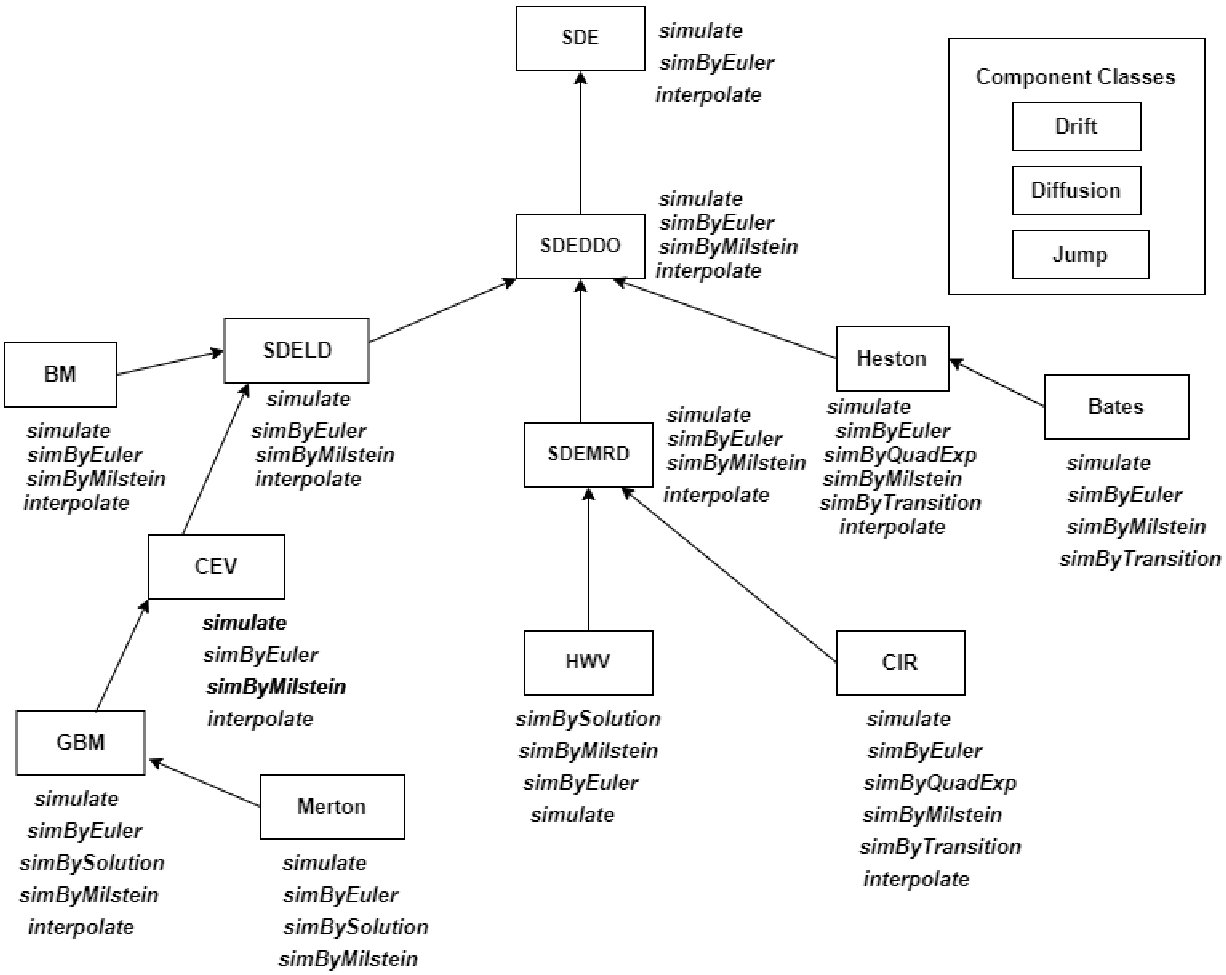
gbm objects display the parameter B as the more familiar Return

More About

Instance Hierarchy

There are inheritance relationships among the SDE classes.

The following figure illustrates the inheritance relationships.



For more information, see “SDE Class Hierarchy” on page 14-5.

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.

Version History

Introduced in R2008a

References

[1] Aït-Sahalia, Yacine. “Testing Continuous-Time Models of the Spot Interest Rate.” *Review of Financial Studies*, vol. 9, no. 2, Apr. 1996, pp. 385–426.

[2] Aït-Sahalia, Yacine. “Transition Densities for Interest Rate and Other Nonlinear Diffusions.” *The Journal of Finance*, vol. 54, no. 4, Aug. 1999, pp. 1361–95.

[3] Glasserman, Paul. *Monte Carlo Methods in Financial Engineering*. Springer, 2004.

[4] Hull, John. *Options, Futures and Other Derivatives*. 7th ed, Prentice Hall, 2009.

[5] Johnson, Norman Lloyd, et al. *Continuous Univariate Distributions*. 2nd ed, Wiley, 1994.

[6] Shreve, Steven E. *Stochastic Calculus for Finance*. Springer, 2004.

See Also

`drift` | `diffusion` | `cev` | `bm` | `simulate` | `interpolate` | `simByEuler` | `nearcorr`

Topics

“Creating Geometric Brownian Motion (GBM) Models” on page 14-22
“Representing Market Models Using SDELD, CEV, and GBM Objects” on page 14-30
“Simulating Equity Prices” on page 14-28
“Simulating Interest Rates” on page 14-48
“Stratified Sampling” on page 14-57
“Price American Basket Options Using Standard Monte Carlo and Quasi-Monte Carlo Simulation” on page 14-70
“Base SDE Models” on page 14-14
“Drift and Diffusion Models” on page 14-16
“Linear Drift Models” on page 14-19
“Parametric Models” on page 14-21
“SDEs” on page 14-2
“SDE Models” on page 14-7
“SDE Class Hierarchy” on page 14-5
“Quasi-Monte Carlo Simulation” on page 14-62
“Performance Considerations” on page 14-64

simBySolution

Simulate approximate solution of diagonal-drift GBM processes

Syntax

```
[Paths,Times,Z] = simBySolution(MDL,NPeriods)
[Paths,Times,Z] = simBySolution( ___ ,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( ___ ,Name,Value)` adds optional name-value pair arguments.

You can perform quasi-Monte Carlo simulations using the name-value arguments for `MonteCarloMethod`, `QuasiSequence` and `BrownianMotionMethod`. For more information, see “Quasi-Monte Carlo Simulation” on page 14-62.

Examples

Simulating Equity Markets Using GBM Simulation Functions

Use GBM simulation functions. Separable GBM models have two specific simulation functions:

- An overloaded Euler simulation function (`simulate`), designed for optimal performance.
- A `simBySolution` function 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 14-28, and the GBM model as in “Representing Market Models Using SDELD, CEV, and GBM Objects” on page 14-30.

```
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);
```

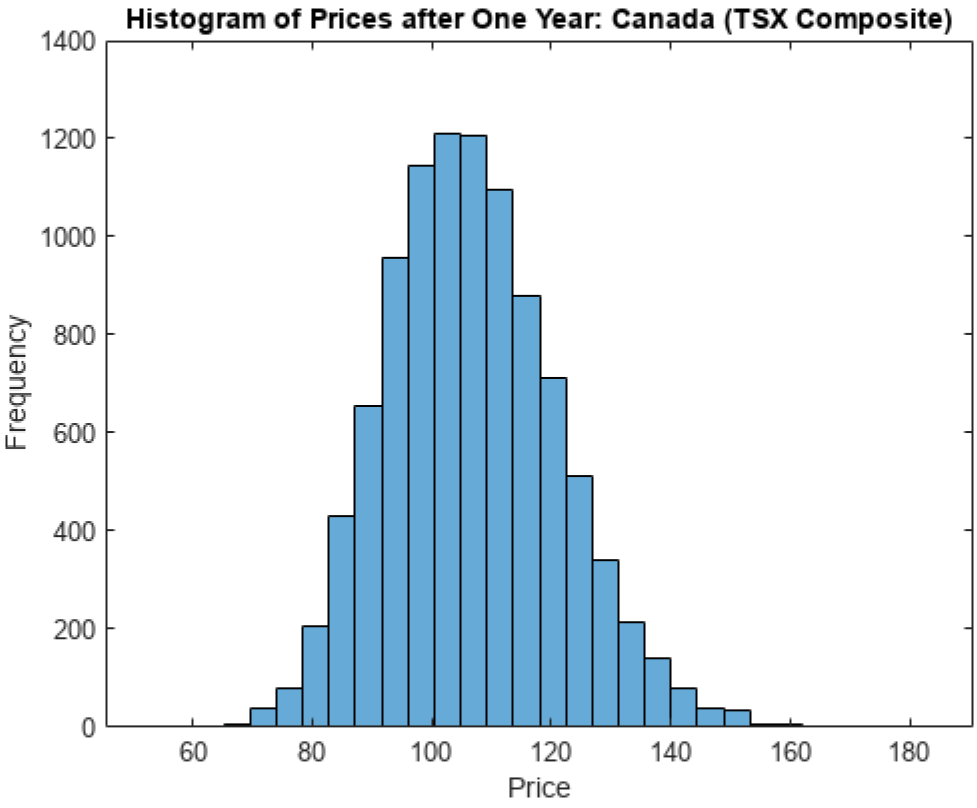
```
SDE = sde(F, G, 'Correlation', ...
    correlation, 'StartState', X);

GBM = gbm(diag(expReturn),diag(sigma), 'Correlation', ...
    correlation, 'StartState', X);
```

To illustrate the performance benefit of the overloaded Euler approximation function (`simulate`), 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
```

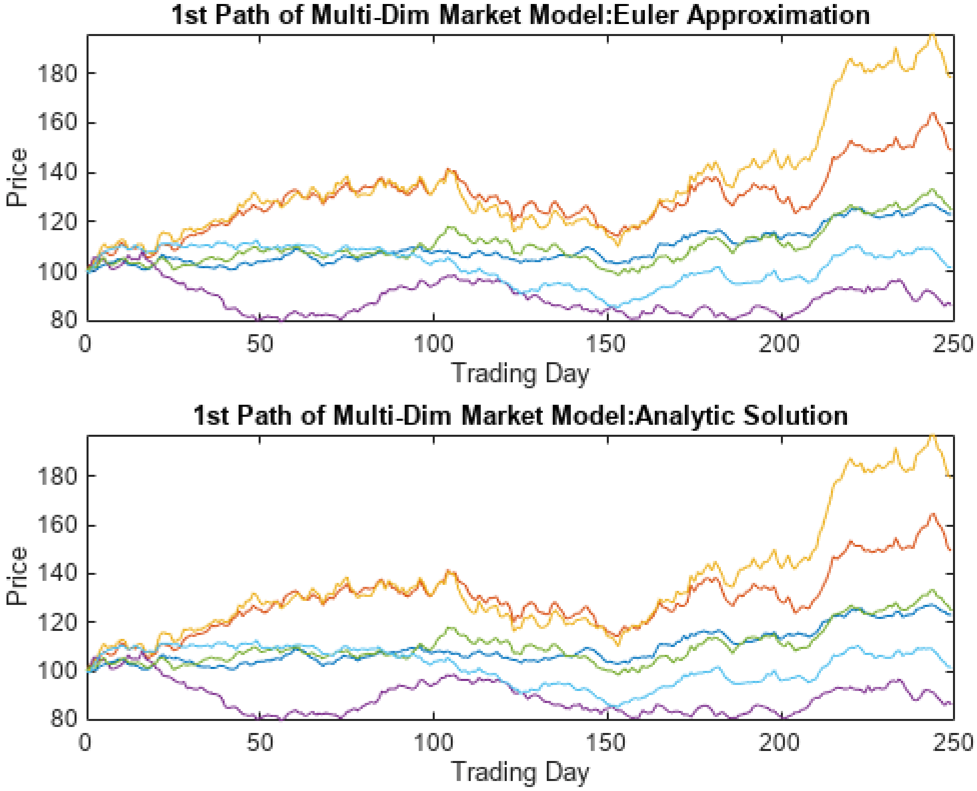
Using this sample size, examine the terminal distribution of Canada's TSX Composite to verify qualitatively the lognormal character of the data.

```
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')
```

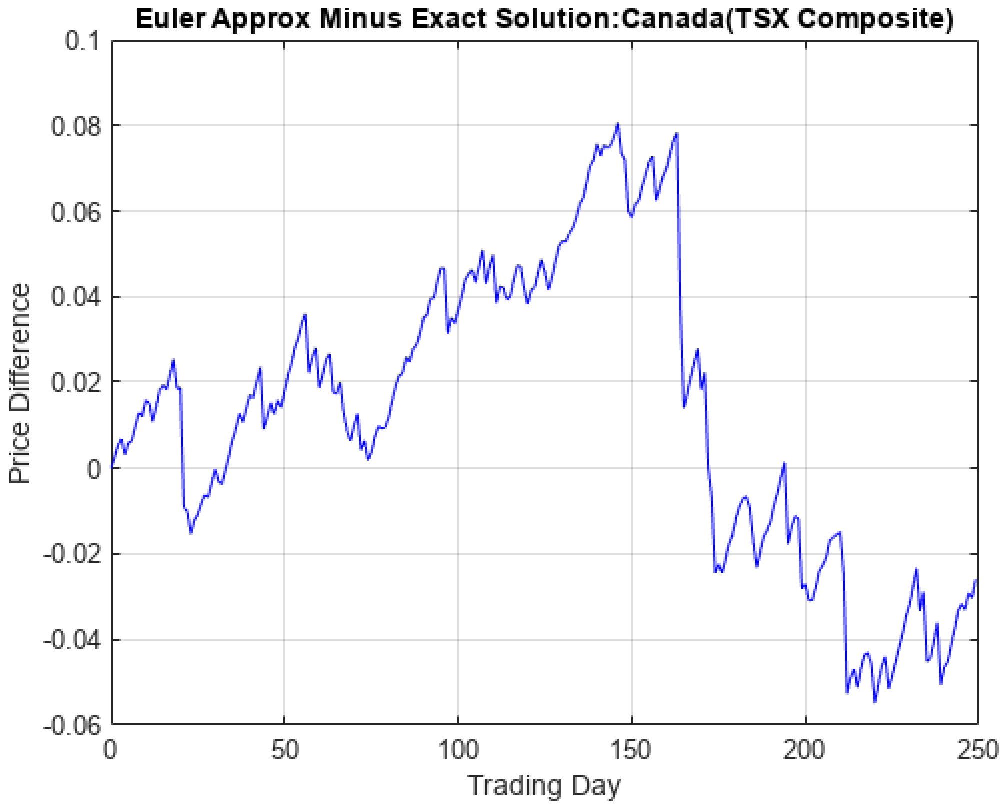

1st Path of Multi-Dim Market Model:Euler Approximation

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 functions:

```
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 function is often the least accurate and most general method available. All models shipped in the simulation suite have the `simByEuler` function.

In contrast, the `simBySolution` function provides a more accurate description of the underlying model. This function 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 14-65.

Simulating Equity Markets Using GBM Model with Quasi-Monte Carlo Simulation

This example shows how to use `simBySolution` with a GBM model to perform a quasi-Monte Carlo simulation. Quasi-Monte Carlo simulation is a Monte Carlo simulation that uses quasi-random sequences instead pseudo random numbers.

Load the `Data_GlobalIdx2` data set and specify the GBM model as in “Representing Market Models Using SDELD, CEV, and GBM Objects” on page 14-30.

```
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);
X           = 100;
X           = X(ones(nVariables,1));

GBM = gbm(diag(expReturn),diag(sigma), 'Correlation', ...
    correlation, 'StartState', X);
```

Perform a quasi-Monte Carlo simulation by using `simBySolution` with the optional name-value arguments for `'MonteCarloMethod'`,`'QuasiSequence'`, and `'BrownianMotionMethod'`.

```
[paths,time,z] = simBySolution(GBM, 10,'ntrials',4096,'MonteCarloMethod','quasi','QuasiSequence'
```

Input Arguments

`MDL` — Geometric Brownian motion (GBM) model

`gbm` object

Geometric Brownian motion (GBM) model, specified as a `gbm` object that is created using `gbm`.

Data Types: `object`

`NPeriods` — Number of simulation periods

positive integer

Number of simulation periods, specified as a positive scalar integer. The value of `NPeriods` determines the number of rows of the simulated output series.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Paths,Times,Z] = simBySolution(GBM,NPeriods,'DeltaTime',dt,'NTrials',10)`

NTrials — Simulated trials (sample paths) of NPERIODS observations each

1 (single path of correlated state variables) (default) | positive integer

Simulated trials (sample paths) of NPERIODS observations each, specified as the comma-separated pair consisting of 'NTrials' and a positive scalar integer.

Data Types: double

DeltaTimes — Positive time increments between observations

1 (default) | scalar | column vector

Positive time increments between observations, specified as the comma-separated pair consisting of 'DeltaTimes' and a scalar or a NPERIODS-by-1 column vector.

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.

Data Types: double

NSteps — Number of intermediate time steps within each time increment *dt* (specified as DeltaTime)

1 (indicating no intermediate evaluation) (default) | positive integer

Number of intermediate time steps within each time increment *dt* (specified as DeltaTime), specified as the comma-separated pair consisting of 'NSteps' and a positive scalar integer.

The simBySolution function 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.

Data Types: double

Antithetic — Flag to indicate whether simBySolution uses antithetic sampling to generate the Gaussian random variates

False (no antithetic sampling) (default) | logical with values True or False

Flag to indicate whether simBySolution uses antithetic sampling to generate the Gaussian random variates that drive the Brownian motion vector (Wiener processes), specified as the comma-separated pair consisting of 'Antithetic' and a scalar logical flag with a value of True or False.

When you specify True, 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.

Note If you specify an input noise process (see Z), simBySolution ignores the value of Antithetic.

Data Types: logical

MonteCarloMethod — Monte Carlo method to simulate stochastic processes

"standard" (default) | string with values "standard", "quasi", or "randomized-quasi" | character vector with values 'standard', 'quasi', or 'randomized-quasi'

Monte Carlo method to simulate stochastic processes, specified as the comma-separated pair consisting of 'MonteCarloMethod' and a string or character vector with one of the following values:

- "standard" — Monte Carlo using pseudo random numbers.
- "quasi" — Quasi-Monte Carlo using low-discrepancy sequences.
- "randomized-quasi" — Randomized quasi-Monte Carlo.

Note If you specify an input noise process (see Z), simBySolution ignores the value of MonteCarloMethod.

Data Types: string | char

QuasiSequence — Low discrepancy sequence to drive the stochastic processes

"sobol" (default) | string with value "sobol" | character vector with value 'sobol'

Low discrepancy sequence to drive the stochastic processes, specified as the comma-separated pair consisting of 'QuasiSequence' and a string or character vector with one of the following values:

- "sobol" — Quasi-random low-discrepancy sequences that use a base of two to form successively finer uniform partitions of the unit interval and then reorder the coordinates in each dimension

Note

- If MonteCarloMethod option is not specified or specified as"standard", QuasiSequence is ignored.
- If you specify an input noise process (see Z), simBySolution ignores the value of QuasiSequence.

Data Types: string | char

BrownianMotionMethod — Brownian motion construction method

"standard" (default) | string with value "brownian-bridge" or "principal-components" | character vector with value 'brownian-bridge' or 'principal-components'

Brownian motion construction method, specified as the comma-separated pair consisting of 'BrownianMotionMethod' and a string or character vector with one of the following values:

- "standard" — The Brownian motion path is found by taking the cumulative sum of the Gaussian variates.
- "brownian-bridge" — The last step of the Brownian motion path is calculated first, followed by any order between steps until all steps have been determined.
- "principal-components" — The Brownian motion path is calculated by minimizing the approximation error.

Note If an input noise process is specified using the Z input argument, BrownianMotionMethod is ignored.

The starting point for a Monte Carlo simulation is the construction of a Brownian motion sample path (or Wiener path). Such paths are built from a set of independent Gaussian variates, using either standard discretization, Brownian-bridge construction, or principal components construction.

Both standard discretization and Brownian-bridge construction share the same variance and therefore the same resulting convergence when used with the MonteCarloMethod using pseudo random numbers. However, the performance differs between the two when the MonteCarloMethod option "quasi" is introduced, with faster convergence seen for "brownian-bridge" construction option and the fastest convergence when using the "principal-components" construction option.

Data Types: string | char

Z — Direct specification of the dependent random noise process used to generate the Brownian motion vector

generates correlated Gaussian variates based on the Correlation member of the SDE object (default) | function | three-dimensional array of dependent random variates

Direct specification of the dependent random noise process used to generate the Brownian motion vector (Wiener process) that drives the simulation, specified as the comma-separated pair consisting of 'Z' and a function or as an (NPERIODS * NSTEPS)-by-NBROWNS-by-NTRIALS three-dimensional array of dependent random variates.

The input argument Z allows you to directly specify the noise generation process. This process takes precedence over the Correlation parameter of the input gbm object and the value of the Antithetic input flag.

Note 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 | function

StorePaths — Flag that indicates how the output array Paths is stored and returned

True (default) | logical with values True or False

Flag that indicates how the output array Paths is stored and returned, specified as the comma-separated pair consisting of 'StorePaths' and a scalar logical flag with a value of True or False.

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` — Sequence of end-of-period processes or state vector adjustments of the form
`simBySolution` makes no adjustments and performs no processing (default) | function | cell array of functions

Sequence of end-of-period processes or state vector adjustments of the form, specified as the comma-separated pair consisting of `'Processes'` and a function or cell array of functions of the form

$$X_t = P(t, X_t)$$

The `simBySolution` function 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.

`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.

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).

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: `cell` | `function`

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 the simulated paths
column vector

Observation times associated with the simulated paths, returned as a `(NPERIODS + 1)`-by-`1` column vector. Each element of `Times` is associated with the corresponding row of `Paths`.

`Z` — Dependent random variates used to generate the Brownian motion vector
array

Dependent random variates used to generate the Brownian motion vector (Wiener processes) that drive the simulation, returned as a `(NPERIODS * NSTEPS)`-by-`NBROWNS`-by-`NTRIALS` three-dimensional time series array.

More About

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.

Algorithms

The `simBySolution` function 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` function 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.

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.

Version History

Introduced in R2008a

R2022a: Perform Quasi-Monte Carlo simulation

Behavior changed in R2022a

Perform Quasi-Monte Carlo simulation using the name-value arguments `MonteCarloMethod` and `QuasiSequence`.

R2022b: Perform Brownian bridge and principal components construction

Behavior changed in R2022b

Perform Brownian bridge and principal components construction using the name-value argument `BrownianMotionMethod`.

References

[1] Aït-Sahalia, Yacine. “Testing Continuous-Time Models of the Spot Interest Rate.” *Review of Financial Studies* 9, no. 2 (Apr. 1996): 385–426.

[2] Aït-Sahalia, Yacine. “Transition Densities for Interest Rate and Other Nonlinear Diffusions.” *The Journal of Finance* 54, no. 4 (Aug. 1999): 1361–95.

[3] Glasserman, Paul. *Monte Carlo Methods in Financial Engineering*. New York: Springer-Verlag, 2004.

[4] Hull, John C. *Options, Futures and Other Derivatives*. 7th ed, Prentice Hall, 2009.

[5] Johnson, Norman Lloyd, Samuel Kotz, and Narayanaswamy Balakrishnan. *Continuous Univariate Distributions*. 2nd ed. Wiley Series in Probability and Mathematical Statistics. New York: Wiley, 1995.

[6] Shreve, Steven E. *Stochastic Calculus for Finance*. New York: Springer-Verlag, 2004.

See Also

`simByEuler` | `simulate` | `gbm` | `simBySolution`

Topics

“Simulating Equity Prices” on page 14-28
“Simulating Interest Rates” on page 14-48
“Stratified Sampling” on page 14-57
“Price American Basket Options Using Standard Monte Carlo and Quasi-Monte Carlo Simulation” on page 14-70
“Base SDE Models” on page 14-14
“Drift and Diffusion Models” on page 14-16
“Linear Drift Models” on page 14-19
“Parametric Models” on page 14-21
“SDEs” on page 14-2
“SDE Models” on page 14-7
“SDE Class Hierarchy” on page 14-5
“Quasi-Monte Carlo Simulation” on page 14-62
“Performance Considerations” on page 14-64

simBySolution

Simulate approximate solution of diagonal-drift `HWV` processes

Syntax

```
[Paths,Times,Z] = simBySolution(MDL,NPeriods)
[Paths,Times,Z] = simBySolution( ___ ,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( ___ ,Name,Value)` adds optional name-value pair arguments.

You can perform quasi-Monte Carlo simulations using the name-value arguments for `MonteCarloMethod`, `QuasiSequence`, and `BrownianMotionMethod`. For more information, see “Quasi-Monte Carlo Simulation” on page 14-62.

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` function 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);
```

Quasi-Monte Carlo Simulation Using HWV Model

This example shows how to use `simBySolution` with a HWV model to perform a quasi-Monte Carlo simulation. Quasi-Monte Carlo simulation is a Monte Carlo simulation that uses quasi-random sequences instead pseudo random numbers.

```
HWV = hwv(1.5,100,10,'startstate',100);
```

Perform a quasi-Monte Carlo simulation by using `simBySolution` with the optional name-value arguments for `'MonteCarloMethod'`,`'QuasiSequence'`, and `'BrownianMotionMethod'`.

```
[paths,time,z] = simBySolution(HWV, 10,'ntrials',4096,'MonteCarloMethod','quasi','QuasiSequence'
```

Input Arguments

`MDL` — Hull-White/Vasicek (HWV) model
hwv object

Hull-White/Vasicek (HWV) mode, specified as a `hwv` object that is created using `hwv`.

Data Types: `object`

`NPeriods` — Number of simulation periods
positive integer

Number of simulation periods, specified as a positive scalar integer. The value of `NPeriods` determines the number of rows of the simulated output series.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Paths,Times,Z] = simBySolution(HWV,NPeriods,'DeltaTime',dt,'NTrials',10)`

`NTrials` — Simulated trials (sample paths) of `NPeriods` observations each
1 (single path of correlated state variables) (default) | positive integer

Simulated trials (sample paths) of `NPeriods` observations each, specified as the comma-separated pair consisting of `'NTrials'` and a positive scalar integer.

Data Types: `double`

`DeltaTimes` — Positive time increments between observations
1 (default) | scalar | column vector

Positive time increments between observations, specified as the comma-separated pair consisting of `'DeltaTimes'` and a scalar or a `NPeriods`-by-`1` column vector.

`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.

Data Types: `double`

`NSteps` — Number of intermediate time steps within each time increment *dt* (specified as `DeltaTime`)

`1` (indicating no intermediate evaluation) (default) | positive integer

Number of intermediate time steps within each time increment *dt* (specified as `DeltaTime`), specified as the comma-separated pair consisting of `'NSteps'` and a positive scalar integer.

The `simBySolution` function 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.

Data Types: `double`

`Antithetic` — Flag to indicate whether `simBySolution` uses antithetic sampling to generate the Gaussian random variates

`False` (no antithetic sampling) (default) | logical with values `True` or `False`

Flag to indicate whether `simBySolution` uses antithetic sampling to generate the Gaussian random variates that drive the Brownian motion vector (Wiener processes), specified as the comma-separated pair consisting of `'Antithetic'` and a scalar logical flag with a value of `True` or `False`.

When you specify `True`, `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.

Note If you specify an input noise process (see `Z`), `simBySolution` ignores the value of `Antithetic`.

Data Types: `logical`

`MonteCarloMethod` — Monte Carlo method to simulate stochastic processes

`"standard"` (default) | string with values `"standard"`, `"quasi"`, or `"randomized-quasi"` | character vector with values `'standard'`, `'quasi'`, or `'randomized-quasi'`

Monte Carlo method to simulate stochastic processes, specified as the comma-separated pair consisting of `'MonteCarloMethod'` and a string or character vector with one of the following values:

- `"standard"` — Monte Carlo using pseudo random numbers.
- `"quasi"` — Quasi-Monte Carlo using low-discrepancy sequences.

- "randomized-quasi" — Randomized quasi-Monte Carlo.

Note If you specify an input noise process (see Z), simBySolution ignores the value of MonteCarloMethod.

Data Types: string | char

QuasiSequence — Low discrepancy sequence to drive the stochastic processes
"sobol" (default) | string with value "sobol" | character vector with value 'sobol'

Low discrepancy sequence to drive the stochastic processes, specified as the comma-separated pair consisting of 'QuasiSequence' and a string or character vector with one of the following values:

- "sobol" — Quasi-random low-discrepancy sequences that use a base of two to form successively finer uniform partitions of the unit interval and then reorder the coordinates in each dimension

Note

- If MonteCarloMethod option is not specified or specified as"standard", QuasiSequence is ignored.
- If you specify an input noise process (see Z), simBySolution ignores the value of QuasiSequence.

Data Types: string | char

BrownianMotionMethod — Brownian motion construction method
"standard" (default) | string with value "brownian-bridge" or "principal-components" | character vector with value 'brownian-bridge' or 'principal-components'

Brownian motion construction method, specified as the comma-separated pair consisting of 'BrownianMotionMethod' and a string or character vector with one of the following values:

- "standard" — The Brownian motion path is found by taking the cumulative sum of the Gaussian variates.
- "brownian-bridge" — The last step of the Brownian motion path is calculated first, followed by any order between steps until all steps have been determined.
- "principal-components" — The Brownian motion path is calculated by minimizing the approximation error.

Note If an input noise process is specified using the Z input argument, BrownianMotionMethod is ignored.

The starting point for a Monte Carlo simulation is the construction of a Brownian motion sample path (or Wiener path). Such paths are built from a set of independent Gaussian variates, using either standard discretization, Brownian-bridge construction, or principal components construction.

Both standard discretization and Brownian-bridge construction share the same variance and therefore the same resulting convergence when used with the MonteCarloMethod using pseudo random numbers. However, the performance differs between the two when the MonteCarloMethod option "quasi" is introduced, with faster convergence seen for "brownian-bridge" construction option and the fastest convergence when using the "principal-components" construction option.

Data Types: string | char

Z — Direct specification of the dependent random noise process used to generate the Brownian motion vector
generates correlated Gaussian variates based on the Correlation member of the SDE object (default) | function | three-dimensional array of dependent random variates

Direct specification of the dependent random noise process used to generate the Brownian motion vector (Wiener process) that drives the simulation, specified as the comma-separated pair consisting of 'Z' and a function or as an (NPeriods * NSteps)-by-NBrowns-by-NTrials three-dimensional array of dependent random variates.

The input argument Z allows you to directly specify the noise generation process. This process takes precedence over the Correlation parameter of the input gbm object and the value of the Antithetic input flag.

Note 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 | function

StorePaths — Flag that indicates how the output array Paths is stored and returned
True (default) | logical with values True or False

Flag that indicates how the output array Paths is stored and returned, specified as the comma-separated pair consisting of 'StorePaths' and a scalar logical flag with a value of True or False.

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 — Sequence of end-of-period processes or state vector adjustments of the form
simBySolution makes no adjustments and performs no processing (default) | function | cell array of functions

Sequence of end-of-period processes or state vector adjustments of the form, specified as the comma-separated pair consisting of 'Processes' and a function or cell array of functions of the form

$$X_t = P(t, X_t)$$

The simBySolution function 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.

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.

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).

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: cell | function

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 the simulated paths

column vector

Observation times associated with the simulated paths, returned as a (NPeriods + 1)-by-1 column vector. Each element of Times is associated with the corresponding row of Paths.

Z — Dependent random variates used to generate the Brownian motion vector

array

Dependent random variates used to generate the Brownian motion vector (Wiener processes) that drive the simulation, returned as a (NPeriods * NSteps)-by-NBrowns-by-NTrials three-dimensional time series array.

More About

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.

Algorithms

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.

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.

Version History

Introduced in R2008a

R2022a: Perform Quasi-Monte Carlo simulation

Behavior changed in R2022a

Perform Quasi-Monte Carlo simulation using the name-value arguments `MonteCarloMethod` and `QuasiSequence`.

R2022b: Perform Brownian bridge and principal components construction
Behavior changed in R2022b

Perform Brownian bridge and principal components construction using the name-value argument `BrownianMotionMethod`.

References

[1] Aït-Sahalia, Yacine. “Testing Continuous-Time Models of the Spot Interest Rate.” *Review of Financial Studies* 9, no. 2 (Apr. 1996): 385–426.

[2] Aït-Sahalia, Yacine. “Transition Densities for Interest Rate and Other Nonlinear Diffusions.” *The Journal of Finance* 54, no. 4 (Aug. 1999): 1361–95.

[3] Glasserman, Paul. *Monte Carlo Methods in Financial Engineering*. New York: Springer-Verlag, 2004.

[4] Hull, John C. *Options, Futures and Other Derivatives*. 7th ed, Prentice Hall, 2009.

[5] Johnson, Norman Lloyd, Samuel Kotz, and Narayanaswamy Balakrishnan. *Continuous Univariate Distributions*. 2nd ed. Wiley Series in Probability and Mathematical Statistics. New York: Wiley, 1995.

[6] Shreve, Steven E. *Stochastic Calculus for Finance*. New York: Springer-Verlag, 2004.

See Also

`simByEuler` | `simulate` | `hwv` | `simBySolution`

Topics

“Creating Hull-White/Vasicek (HWV) Gaussian Diffusion Models” on page 14-25
“Simulating Equity Prices” on page 14-28
“Simulating Interest Rates” on page 14-48
“Stratified Sampling” on page 14-57
“Price American Basket Options Using Standard Monte Carlo and Quasi-Monte Carlo Simulation” on page 14-70
“Base SDE Models” on page 14-14
“Drift and Diffusion Models” on page 14-16
“Linear Drift Models” on page 14-19
“Parametric Models” on page 14-21
“SDEs” on page 14-2
“SDE Models” on page 14-7
“SDE Class Hierarchy” on page 14-5
“Quasi-Monte Carlo Simulation” on page 14-62
“Performance Considerations” on page 14-64

heston

`Heston` model

Description

Creates and displays `heston` objects, which derive from the `sdeddo` (SDE from drift and diffusion objects).

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.

Creation

Syntax

```
heston = heston(Return,Speed,Level,Volatility)
heston = heston( ___ ,Name,Value)
```

Description

`heston = heston(Return,Speed,Level,Volatility)` creates a default `heston` object.

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.

`heston = heston( ___ ,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`

The `heston` object has the following “Properties” on page 15-130:

- `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

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`

`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

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 specifyVolatility 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

Note Although heston does not enforce restrictions on the signs of any of these input arguments, each argument is specified as a positive value.

Properties

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

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, heston applies the same initial value to all state variables on all trials.

If StartState is a 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.

Data Types: double

Correlation — Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes)

NBrowns-by-NBrowns identity matrix representing independent Gaussian processes (default) | positive semidefinite matrix | deterministic function

Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes), specified 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. If Correlation is not a symmetric positive semidefinite matrix, use nearcorr to create a positive semidefinite matrix for a correlation matrix.

A Correlation matrix represents a static condition.

As a deterministic function of time, Correlation allows you to specify a dynamic correlation structure.

Data Types: double

Simulation — User-defined simulation function or SDE simulation method

simulation by Euler approximation (simByEuler) (default) | function | SDE simulation method

User-defined simulation function or SDE simulation method, specified as a function or SDE simulation method.

Data Types: function_handle

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)

This property is read-only.

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 `drift` 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 displayed parameters for a `drift` object 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)`

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)

This property is read-only.

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 `diffusion`:

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

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 displayed parameters for a `diffusion` object 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)`

Data Types: `struct` | `double`

Object Functions

interpolate — Brownian interpolation of stochastic differential equations (SDEs) for SDE, BM, GBM, CEV, CIR, HWV, Heston, SDEDDO, SDELD, or SDEMRD models

simulate — Simulate multivariate stochastic differential equations (SDEs) for SDE, BM, GBM, CEV, CIR, HWV, Heston, SDEDDO, SDELD, SDEMRD, Merton, or Bates models

simByEuler — Euler simulation of stochastic differential equations (SDEs) for SDE, BM, GBM, CEV, CIR, HWV, Heston, SDEDDO, SDELD, or SDEMRD models

simByQuadExp — Simulate Bates, Heston, and CIR sample paths by quadratic-exponential discretization scheme

simByTransition — Simulate Heston sample paths with transition density

simByMilstein — Simulate Heston sample paths by Milstein approximation

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)
```

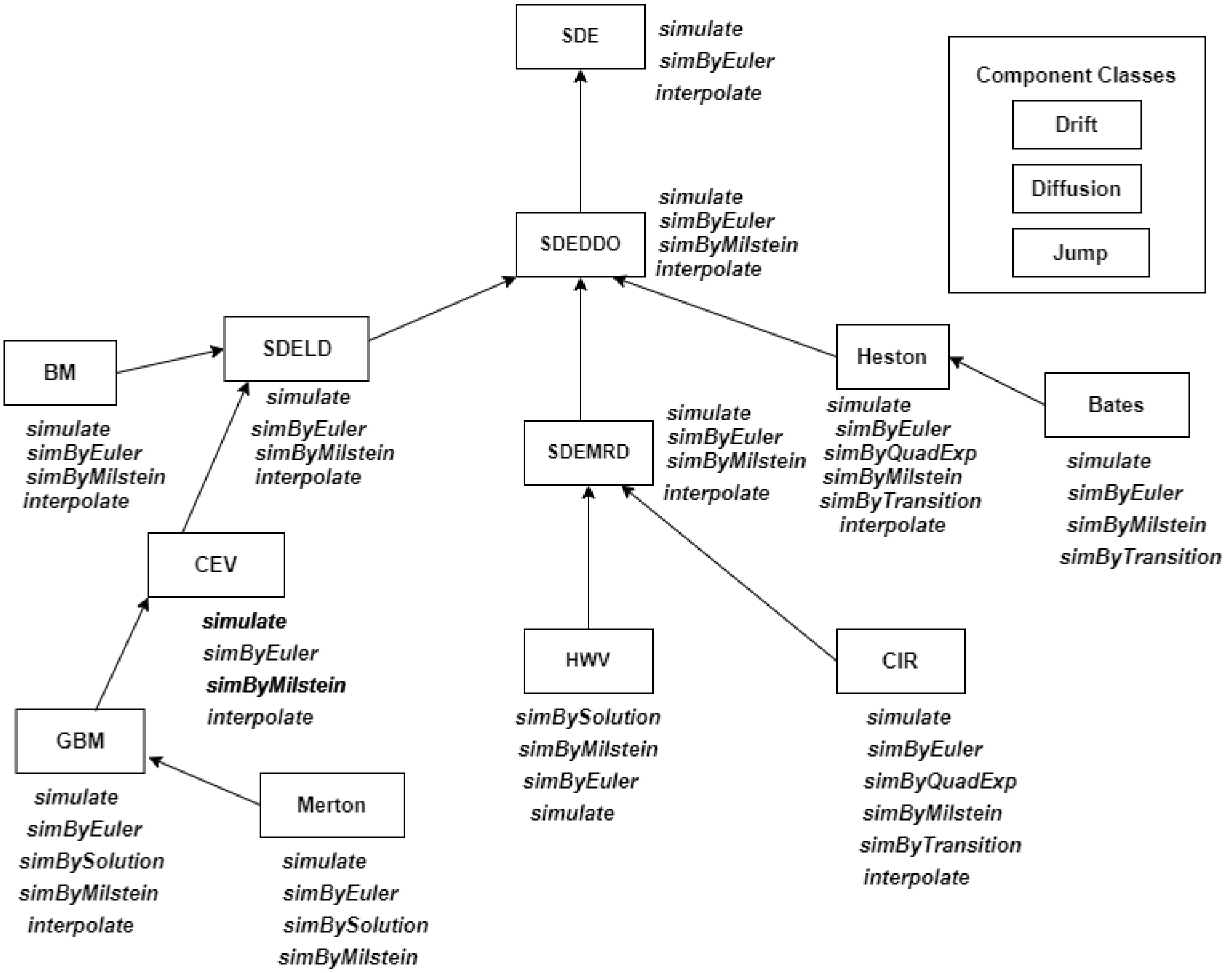
```
obj = 
   Class HESTON: Heston Bivariate Stochastic Volatility
   ----------------------------------------------------
     Dimensions: State = 2, Brownian = 2
   ----------------------------------------------------
      StartTime: 0
     StartState: 1 (2x1 double array) 
    Correlation: 2x2 diagonal 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
```

More About

Instance Hierarchy

There are inheritance relationships among the SDE classes.

The following figure illustrates the inheritance relationships.



For more information, see “SDE Class Hierarchy” on page 14-5.

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.

Version History

Introduced in R2008a

R2023a: Added `simByMilstein` method

Behavior changed in R2023a

Use the `simByMilstein` method to approximate a numerical solution of a stochastic differential equation.

References

[1] Aït-Sahalia, Yacine. “Testing Continuous-Time Models of the Spot Interest Rate.” *Review of Financial Studies*, vol. 9, no. 2, Apr. 1996, pp. 385–426.

[2] Aït-Sahalia, Yacine. “Transition Densities for Interest Rate and Other Nonlinear Diffusions.” *The Journal of Finance*, vol. 54, no. 4, Aug. 1999, pp. 1361–95.

[3] Glasserman, Paul. *Monte Carlo Methods in Financial Engineering*. Springer, 2004.

[4] Hull, John. *Options, Futures and Other Derivatives*. 7th ed, Prentice Hall, 2009.

[5] Johnson, Norman Lloyd, et al. *Continuous Univariate Distributions*. 2nd ed, Wiley, 1994.

[6] Shreve, Steven E. *Stochastic Calculus for Finance*. Springer, 2004.

See Also

`drift` | `diffusion` | `sdeddo` | `simulate` | `interpolate` | `simByEuler` | `nearcorr`

Topics

“Creating Heston Stochastic Volatility Models” on page 14-26
“Simulating Equity Prices” on page 14-28
“Simulating Interest Rates” on page 14-48
“Stratified Sampling” on page 14-57
“Price American Basket Options Using Standard Monte Carlo and Quasi-Monte Carlo Simulation” on page 14-70
“Base SDE Models” on page 14-14
“Drift and Diffusion Models” on page 14-16
“Linear Drift Models” on page 14-19
“Parametric Models” on page 14-21
“SDEs” on page 14-2
“SDE Models” on page 14-7
“SDE Class Hierarchy” on page 14-5
“Quasi-Monte Carlo Simulation” on page 14-62
“Performance Considerations” on page 14-64

hwv

Hull-White/Vasicek (HWV) Gaussian Diffusion model

Description

Create 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 model allows you to simulate vector-valued Hull-White/Vasicek processes of the form:

$$dX_t = S(t)[L(t) - X_t]dt + V(t)dW_t \tag{15-2}$$

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.

Creation

Syntax

```
HWV = hwv(Speed,Level,Sigma)
HWV = hwv( ___ ,Name,Value)
```

Description

HWV = hwv(Speed,Level,Sigma) creates a default HWV object.

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.

`HWV = hwv( ___ ,Name,Value)` creates 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`

The `HWV` object has the following “Properties” on page 15-138:

- `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

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

Note Although the hwv object does not enforce restrictions on the signs of any of these input arguments, each argument is specified as a positive value.

Properties

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

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 hwv object applies the same initial value to all state variables on all trials.

If StartState is a column vector, the hwv object applies a unique initial value to each state variable on all trials.

If StartState is a matrix, the hwv object applies a unique initial value to each state variable on each trial.

Data Types: double

Correlation — Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes)

NBrowns-by-NBrowns identity matrix representing independent Gaussian processes (default) | positive semidefinite matrix | deterministic function

Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes), specified 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. If Correlation is not a symmetric positive semidefinite matrix, use nearcorr to create a positive semidefinite matrix for a correlation matrix.

A Correlation matrix represents a static condition.

As a deterministic function of time, Correlation allows you to specify a dynamic correlation structure.

Data Types: double

Simulation — User-defined simulation function or SDE simulation method

simulation by Euler approximation (simByEuler) (default) | function | SDE simulation method

User-defined simulation function or SDE simulation method, specified as a function or SDE simulation method.

Data Types: function_handle

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)

This property is read-only.

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 drift 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 displayed parameters for a drift object 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)`

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)

This property is read-only.

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 diffusion:

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

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 displayed parameters for a diffusion object 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)`

Data Types: `struct` | `double`

Object Functions

interpolate	Brownian interpolation of stochastic differential equations (SDEs) for SDE, BM, GBM, CEV, CIR, HWV, Heston, SDEDDO, SDELD, or SDEMRD models
simulate	Simulate multivariate stochastic differential equations (SDEs) for SDE, BM, GBM, CEV, CIR, HWV, Heston, SDEDDO, SDELD, SDEMRD, Merton, or Bates models
simByEuler	Euler simulation of stochastic differential equations (SDEs) for SDE, BM, GBM, CEV, CIR, HWV, Heston, SDEDDO, SDELD, or SDEMRD models
simBySolution	Simulate approximate solution of diagonal-drift HWV processes
simByMilstein	Simulate BM, GBM, CEV, HWV, SDEDDO, SDELD, SDEMRD sample paths by Milstein approximation

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 an `hwv` object to represent the model: $dX_t = 0.2(0.1 - X_t)dt + 0.05dW_t$.

```
obj = hwv(0.2, 0.1, 0.05)  % (Speed, Level, Sigma)
```

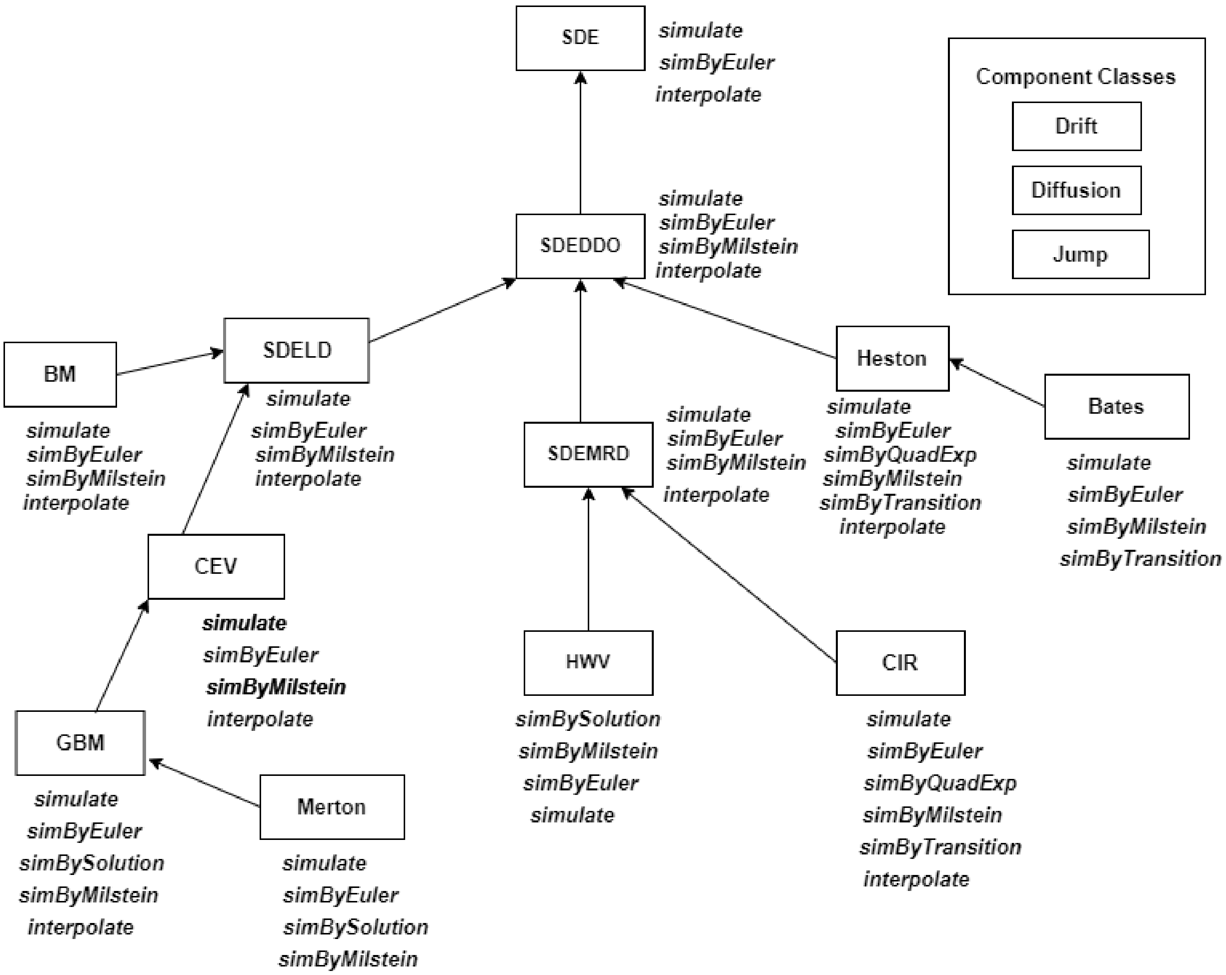
```
obj = 
   Class HWV: Hull-White/Vasicek
   ----------------------------------------
     Dimensions: State = 1, Brownian = 1
   ----------------------------------------
      StartTime: 0
     StartState: 1
    Correlation: 1
          Drift: drift rate function F(t,X(t)) 
      Diffusion: diffusion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

More About

Instance Hierarchy

There are inheritance relationships among the SDE classes.

The following figure illustrates the inheritance relationships.



For more information, see “SDE Class Hierarchy” on page 14-5.

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.

Version History

Introduced in R2008a

References

[1] Aït-Sahalia, Yacine. “Testing Continuous-Time Models of the Spot Interest Rate.” *Review of Financial Studies*, vol. 9, no. 2, Apr. 1996, pp. 385–426.

[2] Aït-Sahalia, Yacine. “Transition Densities for Interest Rate and Other Nonlinear Diffusions.” *The Journal of Finance*, vol. 54, no. 4, Aug. 1999, pp. 1361–95.

[3] Glasserman, Paul. *Monte Carlo Methods in Financial Engineering*. Springer, 2004.

[4] Hull, John. *Options, Futures and Other Derivatives*. 7th ed, Prentice Hall, 2009.

[5] Johnson, Norman Lloyd, et al. *Continuous Univariate Distributions*. 2nd ed, Wiley, 1994.

[6] Shreve, Steven E. *Stochastic Calculus for Finance*. Springer, 2004.

See Also

`drift` | `diffusion` | `sdeddo` | `simulate` | `interpolate` | `simByEuler` | `nearcorr`

Topics

“Creating Hull-White/Vasicek (HWV) Gaussian Diffusion Models” on page 14-25
“Simulating Equity Prices” on page 14-28
“Simulating Interest Rates” on page 14-48
“Stratified Sampling” on page 14-57
“Price American Basket Options Using Standard Monte Carlo and Quasi-Monte Carlo Simulation” on page 14-70
“Base SDE Models” on page 14-14
“Drift and Diffusion Models” on page 14-16
“Linear Drift Models” on page 14-19
“Parametric Models” on page 14-21
“SDEs” on page 14-2
“SDE Models” on page 14-7
“SDE Class Hierarchy” on page 14-5
“Quasi-Monte Carlo Simulation” on page 14-62
“Performance Considerations” on page 14-64

interpolate

Brownian interpolation of stochastic differential equations (SDEs) for `SDE`, `BM`, `GBM`, `CEV`, `CIR`, `HWV`, `Heston`, `SDEDDO`, `SDELD`, or `SDEMRD` models

Syntax

```
[XT,T] = interpolate(MDL,Times,Paths)
[XT,T] = interpolate( ___ ,Name,Value)
```

Description

`[XT,T] = interpolate(MDL,Times,Paths)` performs a Brownian interpolation into a user-specified time series array, based on a piecewise-constant Euler sampling approach.

`[XT,T] = interpolate( ___ ,Name,Value)` adds optional name-value pair arguments.

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);
```

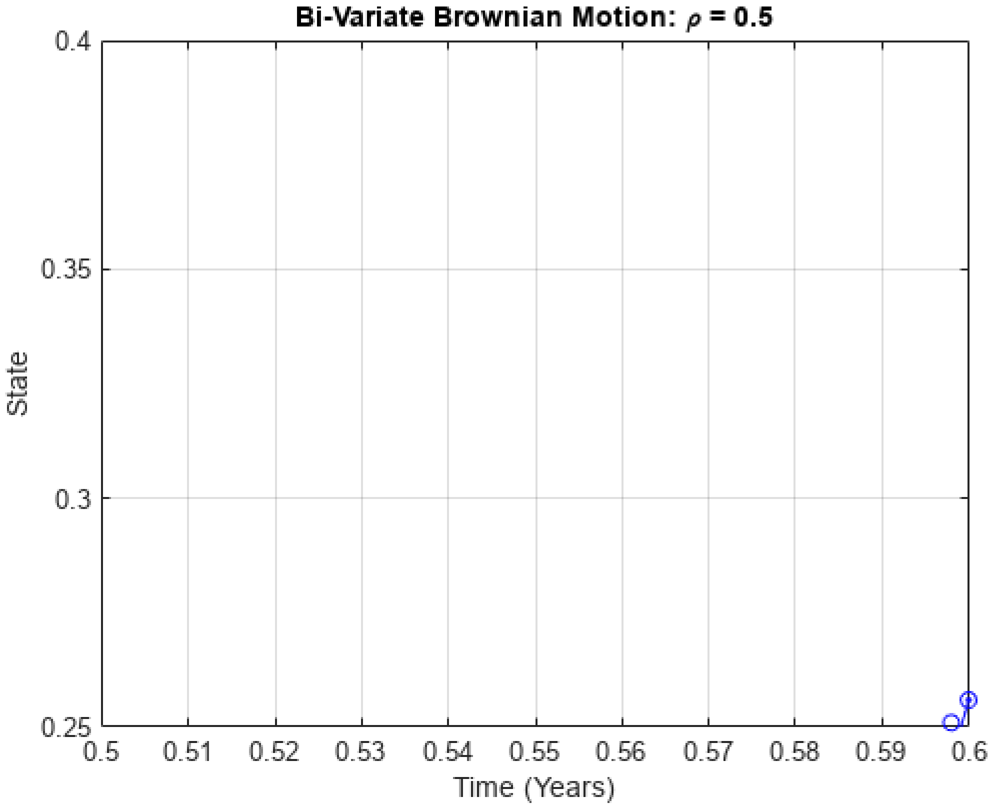
3 It is helpful to examine a small interval in detail.

   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
```

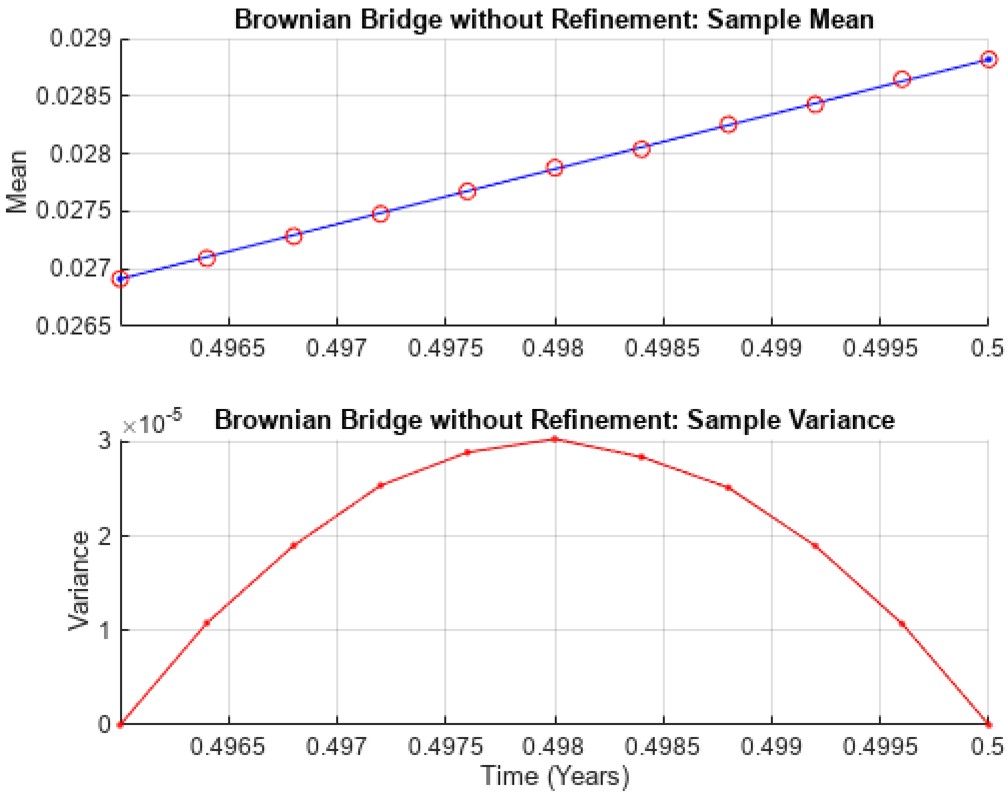
2 In each subinterval, take 25000 independent draws from a Gaussian distribution, conditioned on the simulated states to the left, and right:

```
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.

Input Arguments

`MDL` — Stochastic differential equation model

object

Stochastic differential equation model, specified as an `sde`, `bm`, `gbm`, `cev`, `cir`, `hwv`, `heston`, `sdeddo`, `sdeld`, or `sdemrd` object.

All `MDL` parameters are assumed piecewise constant, evaluated 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.

Data Types: `object`

`Times` — Interpolation times
vector

Interpolation times, specified as a `NTimes` element vector. The length of this vector determines the number of rows in the interpolated output time series `XT`.

Data Types: `double`

`Paths` — Sample paths of correlated state variables
time series array

Sample paths of correlated state variables, specified as a `NPeriods`-by-`NVars`-by-`NTrials` time series array.

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 the `interpolate` function performs the Brownian interpolation.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[XT,T] = interpolate(MDL,T,Paths,'Times',t)`

`Times` — Observation times associated with the time series input `Paths`
zero-based, unit-increment column vector of length `NPeriods` (default) | column vector

Observation times associated with the time series input `Paths`, specified as the comma-separated pair consisting of `'Times'` and a column vector.

Data Types: `double`

`Refine` — Flag that indicates whether `interpolate` uses the interpolation times you request
`False` (`interpolate` bases the interpolation only on the state information specified in `Paths`) (default) | logical with values `True` or `False`

Flag that indicates whether `interpolate` uses the interpolation times you request (see `T`) to refine the interpolation as new information becomes available, specified as the comma-separated pair consisting of `'Refine'` and a logical with a value of `True` or `False`.

Data Types: `logical`

Processes — Sequence of background processes or state vector adjustments
interpolate makes no adjustments and performs no processing (default) | function | cell array of functions

Sequence of background processes or state vector adjustments, specified as the comma-separated pair consisting of 'Processes' and a function or cell array of functions of the form

$$X_t = P(t, X_t)$$

The interpolate function 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.

Data Types: cell | function

Output Arguments

XT — Interpolated state variables
array

Interpolated state variables, returned as a NTimes-by-NVars-by-NTrials time series array.

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 — Interpolation times associated with the output time series XT
column vector

Interpolation times associated with the output time series XT, returned as a NTimes-by-1 column vector.

If the input interpolation time vector Times contains no missing observations (NaNs), the output of T is the same time vector as Times, but with the NaNs removed. This reduces the length of T and the number of rows of XT.

Algorithms

This function 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 function 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 interpolation function does not support user-specified noise processes.

- The interpolate function 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 NaNs).
- The interpolate function assumes that the user-specified time series array Paths is associated with the sde object. For example, the Times and Paths input pair are 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 and Paths input series to take precedence.

Version History

Introduced in R2008a

References

[1] Ait-Sahalia, Y. “Testing Continuous-Time Models of the Spot Interest Rate.” *The Review of Financial Studies*, Spring 1996, Vol. 9, No. 2, pp. 385–426.

[2] Ait-Sahalia, Y. “Transition Densities for Interest Rate and Other Nonlinear Diffusions.” *The Journal of Finance*, Vol. 54, No. 4, August 1999.

[3] Glasserman, P. *Monte Carlo Methods in Financial Engineering.* New York, Springer-Verlag, 2004.

[4] Hull, J. C. *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

[5] Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions.* Vol. 2, 2nd ed. New York, John Wiley & Sons, 1995.

[6] 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 14-28

“Simulating Interest Rates” on page 14-48
“Stratified Sampling” on page 14-57
“Price American Basket Options Using Standard Monte Carlo and Quasi-Monte Carlo Simulation” on page 14-70
“Base SDE Models” on page 14-14
“Drift and Diffusion Models” on page 14-16
“Linear Drift Models” on page 14-19
“Parametric Models” on page 14-21
“SDEs” on page 14-2
“SDE Models” on page 14-7
“SDE Class Hierarchy” on page 14-5
“Performance Considerations” on page 14-64

sde

Stochastic Differential Equation (SDE) model

Description

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.

Creation

Syntax

```
SDE = sde(DriftRate,DiffusionRate)
SDE = sde( ___ ,Name,Value)
```

Description

`SDE = sde(DriftRate,DiffusionRate)` creates a default `SDE` object.

`SDE = sde( ___ ,Name,Value)` creates 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`.

The `SDE` object has the following “Properties” on page 15-154:

- `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

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` | `object`

`DiffusionRate` — `DiffusionRate` is a user-defined diffusion-rate function and represents the parameter *G*

matrix or object of class `diffusion`

`DiffusionRate` is a user-defined diffusion-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` | `object`

Properties

`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

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, 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.

Data Types: double

Correlation — Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes)

NBROWNS-by-NBROWNS identity matrix representing independent Gaussian processes (default) | positive semidefinite matrix | deterministic function

Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes), specified 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. If Correlation is not a symmetric positive semidefinite matrix, use nearcorr to create a positive semidefinite matrix for a correlation matrix.

A Correlation matrix represents a static condition.

As a deterministic function of time, Correlation allows you to specify a dynamic correlation structure.

Data Types: double

Simulation — User-defined simulation function or SDE simulation method

simulation by Euler approximation (simByEuler) (default) | function | SDE simulation method

User-defined simulation function or SDE simulation method, specified as a function or SDE simulation method.

Data Types: function_handle

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)

This property is read-only.

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 drift) 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 displayed parameters for a drift object 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)`

Data Types: object

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)

This property is read-only.

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 diffusion):

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

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 displayed parameters for a diffusion object 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)`

Data Types: `object`

Object Functions

interpolate Brownian interpolation of stochastic differential equations (SDEs) for SDE, BM, GBM, CEV, CIR, HWV, Heston, SDEDDO, SDELD, or SDEMRD models

simulate Simulate multivariate stochastic differential equations (SDEs) for SDE, BM, GBM, CEV, CIR, HWV, Heston, SDEDDO, SDELD, SDEMRD, Merton, or Bates models

simByEuler Euler simulation of stochastic differential equations (SDEs) for SDE, BM, GBM, CEV, CIR, HWV, Heston, SDEDDO, SDELD, or SDEMRD models

Examples

Create an SDE Object

Construct an `SDE` object `obj` to represent a univariate geometric Brownian Motion model of the form: $dX_t = 0.1X_tdt + 0.3X_tdW_t$.

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 `sde` 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.

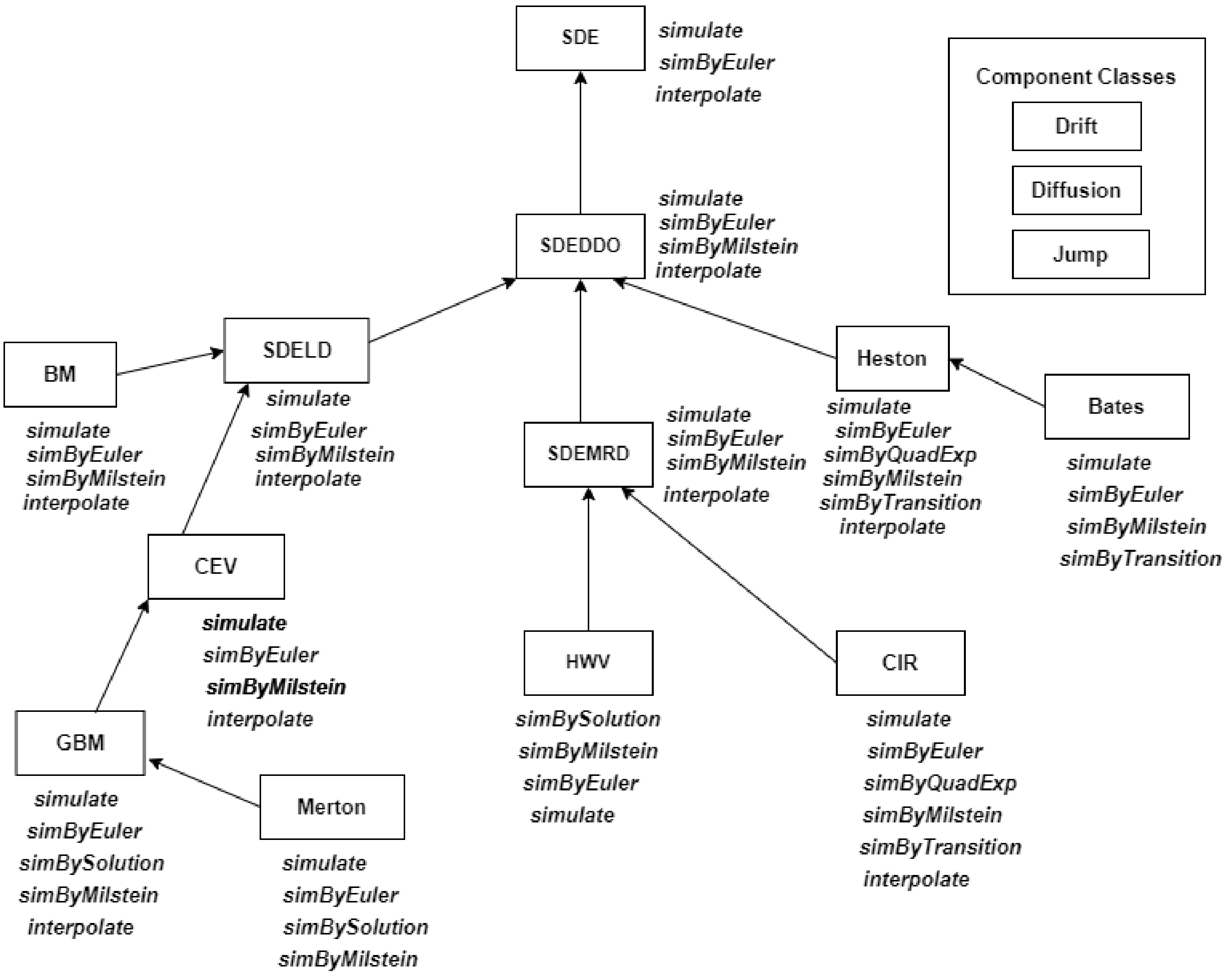
The only exception to the (t, X_t) evaluation interface is Correlation. Specifically, when you enter Correlation as a function, the SDE engine assumes that it is a deterministic function of time, $C(t)$. This restriction on Correlation as a deterministic function of time allows Cholesky factors to be 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.

More About

Instance Hierarchy

There are inheritance relationships among the SDE classes.

The following figure illustrates the inheritance relationships.



For more information, see “SDE Class Hierarchy” on page 14-5.

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.

Version History

Introduced in R2008a

References

[1] Aït-Sahalia, Yacine. “Testing Continuous-Time Models of the Spot Interest Rate.” *Review of Financial Studies*, vol. 9, no. 2, Apr. 1996, pp. 385–426.

[2] Aït-Sahalia, Yacine. “Transition Densities for Interest Rate and Other Nonlinear Diffusions.” *The Journal of Finance*, vol. 54, no. 4, Aug. 1999, pp. 1361–95.

[3] Glasserman, Paul. *Monte Carlo Methods in Financial Engineering*. Springer, 2004.

[4] Hull, John. *Options, Futures and Other Derivatives*. 7th ed, Prentice Hall, 2009.

[5] Johnson, Norman Lloyd, et al. *Continuous Univariate Distributions*. 2nd ed, Wiley, 1994.

[6] Shreve, Steven E. *Stochastic Calculus for Finance*. Springer, 2004.

See Also

`drift` | `diffusion` | `nearcorr`

Topics

“Base SDE Models” on page 14-14
Representing Market Models Using SDE Objects on page 14-28
“Simulating Equity Prices” on page 14-28
“Simulating Interest Rates” on page 14-48
“Stratified Sampling” on page 14-57
“Price American Basket Options Using Standard Monte Carlo and Quasi-Monte Carlo Simulation” on page 14-70
“Drift and Diffusion Models” on page 14-16
“Linear Drift Models” on page 14-19
“Parametric Models” on page 14-21
“SDEs” on page 14-2
“SDE Models” on page 14-7
“SDE Class Hierarchy” on page 14-5
“Quasi-Monte Carlo Simulation” on page 14-62
“Performance Considerations” on page 14-64

sdeddo

Stochastic Differential Equation (SDEDDO) model from Drift and Diffusion components

Description

Creates and displays sdeddo objects, instantiated with objects of classdrift 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 SDEDDO of the form:

$$dX_t = F(t, X_t)dt + G(t, X_t)dW_t \tag{15-3}$$

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.

Creation

Syntax

```
SDEDDO = sdeddo(DriftRate,DiffusionRate)
SDEDDO = sdeddo( ___ ,Name,Value)
```

Description

SDEDDO = sdeddo(DriftRate,DiffusionRate) creates a default SDEDDO object.

SDEDDO = sdeddo(___ ,Name,Value) creates 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.

The SDEDDO object has the following displayed “Properties” on page 15-163:

- 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

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 diffusion-rate function and represents the parameter *G*

matrix or object of class `Diffusion`

`DiffusionRate` is a user-defined diffusion-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`

Properties

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

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, 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.

Data Types: double

Correlation — Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes)
NBrowns-by-NBrowns identity matrix representing independent Gaussian processes (default) | positive semidefinite matrix | deterministic function

Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes), specified 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. If Correlation is not a symmetric positive semidefinite matrix, use nearcorr to create a positive semidefinite matrix for a correlation matrix.

A Correlation matrix represents a static condition.

As a deterministic function of time, Correlation allows you to specify a dynamic correlation structure.

Data Types: double

Simulation — User-defined simulation function or SDE simulation method
simulation by Euler approximation (simByEuler) (default) | function | SDE simulation method

User-defined simulation function or SDE simulation method, specified as a function or SDE simulation method.

Data Types: function_handle

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)

This property is read-only.

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 `drift` 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 displayed parameters for a `drift` object 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)`

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)

This property is read-only.

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 `diffusion`:

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

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 displayed parameters for a `diffusion` object 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)`

Data Types: `struct` | `double`

Object Functions

interpolate	Brownian interpolation of stochastic differential equations (SDEs) for SDE, BM, GBM, CEV, CIR, HWV, Heston, SDEDDO, SDELD, or SDEMRD models
simulate	Simulate multivariate stochastic differential equations (SDEs) for SDE, BM, GBM, CEV, CIR, HWV, Heston, SDEDDO, SDELD, SDEMRD, Merton, or Bates models
simByEuler	Euler simulation of stochastic differential equations (SDEs) for SDE, BM, GBM, CEV, CIR, HWV, Heston, SDEDDO, SDELD, or SDEMRD models
simByMilstein	Simulate BM, GBM, CEV, HWV, SDEDDO, SDELD, SDEMRD sample paths by Milstein approximation

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` function.

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` function 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
```

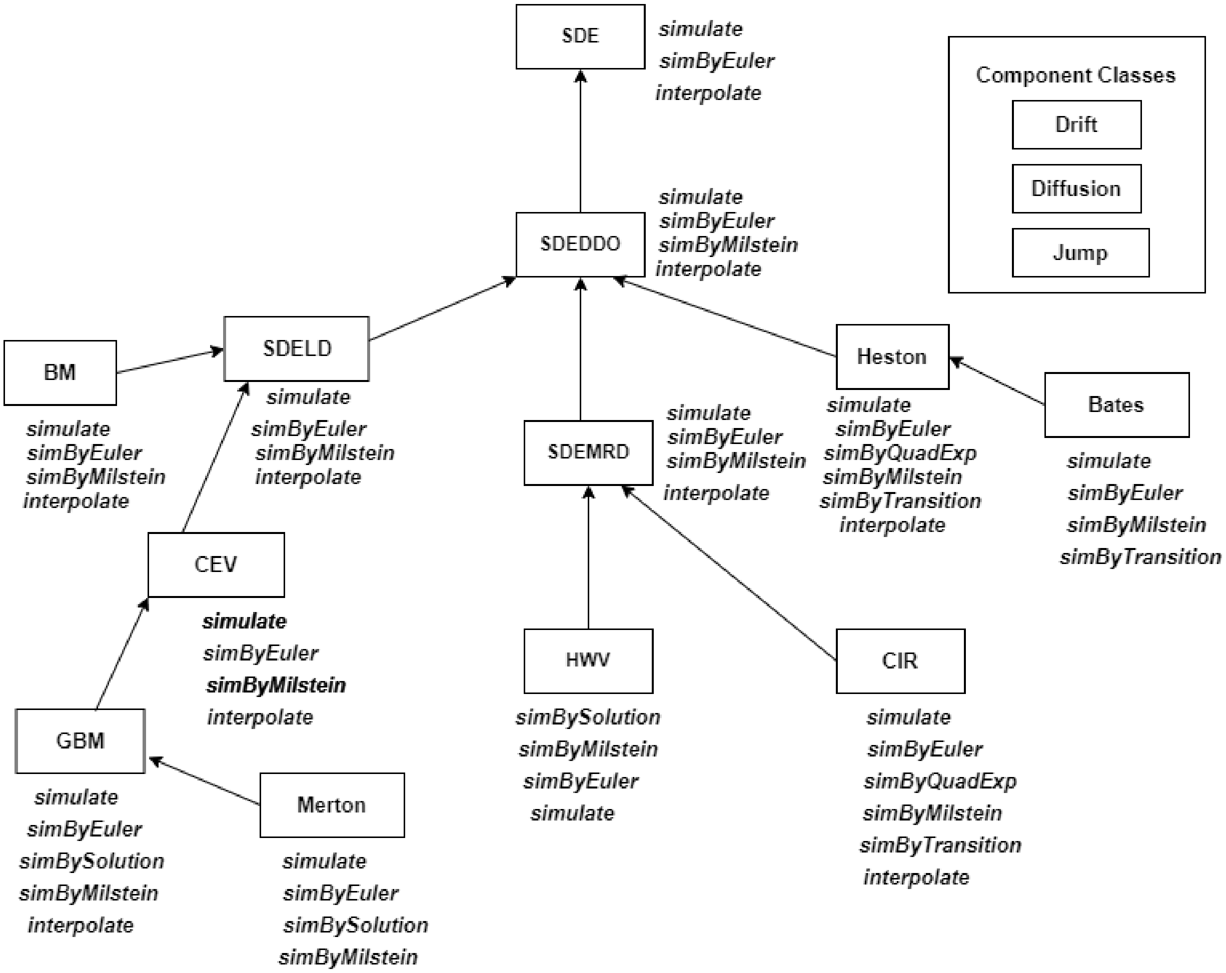
In this example, the object displays the additional parameters associated with input `drift` and `diffusion` objects.

More About

Instance Hierarchy

There are inheritance relationships among the SDE classes.

The following figure illustrates the inheritance relationships.



For more information, see “SDE Class Hierarchy” on page 14-5.

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.

Version History

Introduced in R2008a

References

[1] Aït-Sahalia, Yacine. “Testing Continuous-Time Models of the Spot Interest Rate.” *Review of Financial Studies*, vol. 9, no. 2, Apr. 1996, pp. 385–426.

[2] Aït-Sahalia, Yacine. “Transition Densities for Interest Rate and Other Nonlinear Diffusions.” *The Journal of Finance*, vol. 54, no. 4, Aug. 1999, pp. 1361–95.

[3] Glasserman, Paul. *Monte Carlo Methods in Financial Engineering*. Springer, 2004.

[4] Hull, John. *Options, Futures and Other Derivatives*. 7th ed, Prentice Hall, 2009.

[5] Johnson, Norman Lloyd, et al. *Continuous Univariate Distributions*. 2nd ed, Wiley, 1994.

[6] Shreve, Steven E. *Stochastic Calculus for Finance*. Springer, 2004.

See Also

`drift` | `diffusion` | `sdeld` | `simulate` | `interpolate` | `simByEuler` | `nearcorr`

Topics

“Drift and Diffusion Models” on page 14-16
“Representing Market Models Using SDEDDO Objects” on page 14-29
“Representing Market Models Using SDE Objects” on page 14-28
“Simulating Equity Prices” on page 14-28
“Simulating Interest Rates” on page 14-48
“Stratified Sampling” on page 14-57
“Price American Basket Options Using Standard Monte Carlo and Quasi-Monte Carlo Simulation” on page 14-70
“Base SDE Models” on page 14-14
“Drift and Diffusion Models” on page 14-16
“Linear Drift Models” on page 14-19
“Parametric Models” on page 14-21
“SDEs” on page 14-2
“SDE Models” on page 14-7
“SDE Class Hierarchy” on page 14-5
“Quasi-Monte Carlo Simulation” on page 14-62
“Performance Considerations” on page 14-64

sdeld

SDE with Linear Drift (SDELD) model

Description

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 SDELD 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.

Creation

Syntax

```
SDELD = sdeld(A,B,Alpha,Sigma)
SDELD = sdeld( ___ ,Name,Value)
```

Description

SDELD = sdeld(A,B,Alpha,Sigma) creates a default SDELD object.

SDELD = sdeld(___ ,Name,Value) creates 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.

The SDELD object has the following displayed "Properties" on page 15-171:

- 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

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 thegbm constructor enforces no restrictions on the sign of Sigma volatilities, they are specified as positive values.

Data Types: double | function_handle

Note Although sdeld does not enforce restrictions on the signs of Alpha or Sigma, each parameter is specified as a positive value.

Properties

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

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, 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.

Data Types: double

Correlation — Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes)
NBrowns-by-NBrowns identity matrix representing independent Gaussian processes (default) | positive semidefinite matrix | deterministic function

Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes), specified 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. If Correlation is not a symmetric positive semidefinite matrix, use nearcorr to create a positive semidefinite matrix for a correlation matrix.

A Correlation matrix represents a static condition.

As a deterministic function of time, Correlation allows you to specify a dynamic correlation structure.

Data Types: double

Simulation — User-defined simulation function or SDE simulation method
simulation by Euler approximation (simByEuler) (default) | function | SDE simulation method

User-defined simulation function or SDE simulation method, specified as a function or SDE simulation method.

Data Types: function_handle

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)

This property is read-only.

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 drift 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 displayed parameters for a drift object 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)`

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)

This property is read-only.

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 `diffusion`:

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

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 displayed parameters for a `diffusion` object 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)`

Data Types: `struct` | `double`

Object Functions

interpolate	Brownian interpolation of stochastic differential equations (SDEs) for SDE, BM, GBM, CEV, CIR, HWV, Heston, SDEDDO, SDELD, or SDEMRD models
simulate	Simulate multivariate stochastic differential equations (SDEs) for SDE, BM, GBM, CEV, CIR, HWV, Heston, SDEDDO, SDELD, SDEMRD, Merton, or Bates models
simByEuler	Euler simulation of stochastic differential equations (SDEs) for SDE, BM, GBM, CEV, CIR, HWV, Heston, SDEDDO, SDELD, or SDEMRD models
simByMilstein	Simulate BM, GBM, CEV, HWV, SDEDDO, SDELD, SDEMRD sample paths by Milstein approximation

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
```

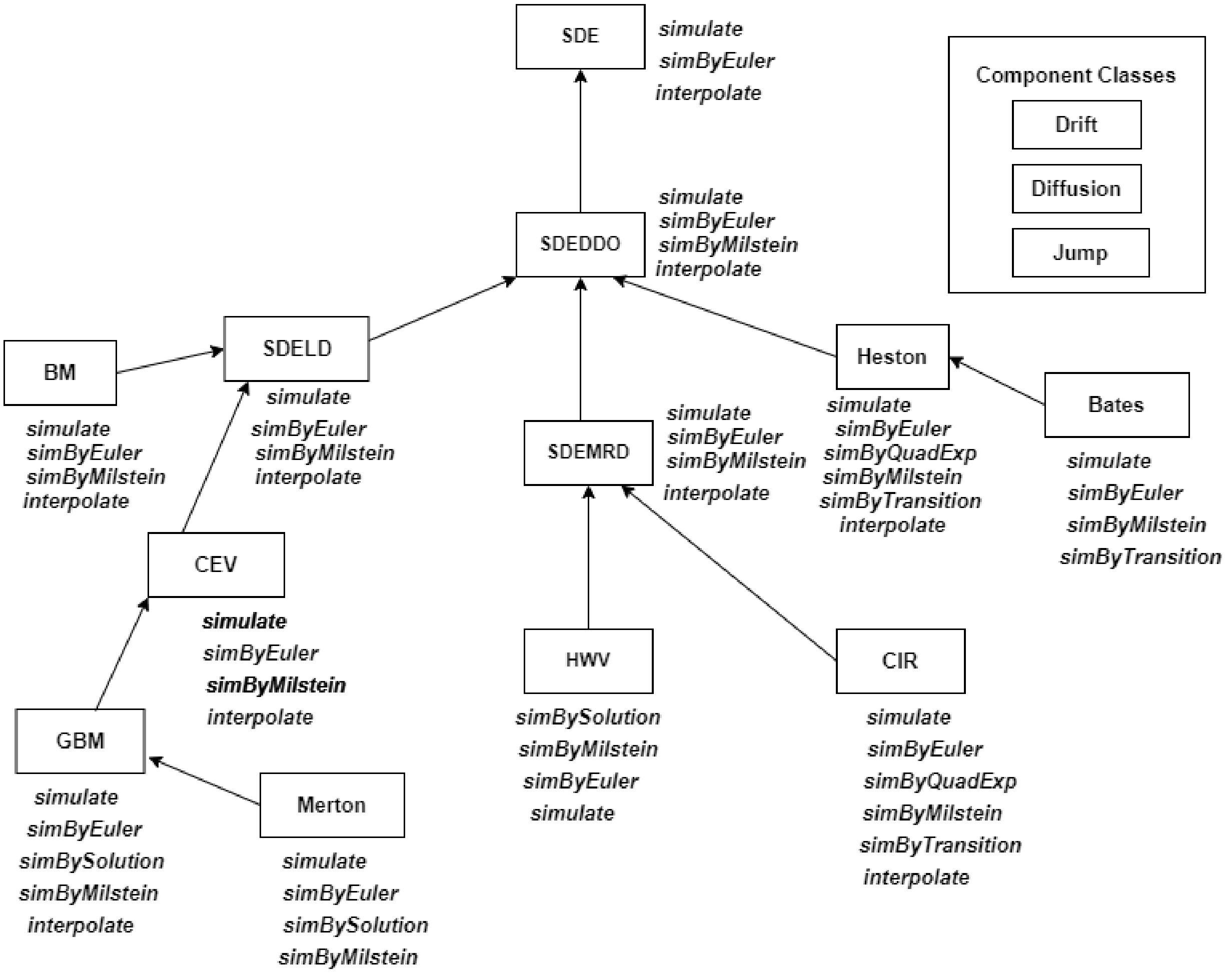
`sdeld` objects provide a parametric alternative to the mean-reverting drift form and also provide an alternative interface to the `sdeddo` parent class, because you can create an object without first having to create its drift and diffusion-rate components.

More About

Instance Hierarchy

There are inheritance relationships among the SDE classes.

The following figure illustrates the inheritance relationships.



For more information, see “SDE Class Hierarchy” on page 14-5.

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.

Version History

Introduced in R2008a

References

[1] Aït-Sahalia, Yacine. “Testing Continuous-Time Models of the Spot Interest Rate.” *Review of Financial Studies*, vol. 9, no. 2, Apr. 1996, pp. 385–426.

[2] Aït-Sahalia, Yacine. “Transition Densities for Interest Rate and Other Nonlinear Diffusions.” *The Journal of Finance*, vol. 54, no. 4, Aug. 1999, pp. 1361–95.

[3] Glasserman, Paul. *Monte Carlo Methods in Financial Engineering*. Springer, 2004.

[4] Hull, John. *Options, Futures and Other Derivatives*. 7th ed, Prentice Hall, 2009.

[5] Johnson, Norman Lloyd, et al. *Continuous Univariate Distributions*. 2nd ed, Wiley, 1994.

[6] Shreve, Steven E. *Stochastic Calculus for Finance*. Springer, 2004.

See Also

`drift` | `diffusion` | `sdeddo` | `simByEuler` | `nearcorr`

Topics

“Linear Drift Models” on page 14-19
Implementing Multidimensional Equity Market Models, Implementation 3: Using SDELD, CEV, and GBM Objects on page 14-30
“Simulating Equity Prices” on page 14-28
“Simulating Interest Rates” on page 14-48
“Stratified Sampling” on page 14-57
“Price American Basket Options Using Standard Monte Carlo and Quasi-Monte Carlo Simulation” on page 14-70
“Base SDE Models” on page 14-14
“Drift and Diffusion Models” on page 14-16
“Linear Drift Models” on page 14-19
“Parametric Models” on page 14-21
“SDEs” on page 14-2
“SDE Models” on page 14-7
“SDE Class Hierarchy” on page 14-5
“Quasi-Monte Carlo Simulation” on page 14-62
“Performance Considerations” on page 14-64

sdemrd

SDE with Mean-Reverting Drift (SDEMRD) model

Description

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 SDEMRD 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.

Creation

Syntax

```
SDEMRD = sdemrd(Speed,Level,Alpha,Sigma)
SDEMRD = sdemrd(___,Name,Value)
```

Description

SDEMRD = sdemrd(Speed,Level,Alpha,Sigma) creates a default SDEMRD object.

SDEMRD = sdemrd(___,Name,Value) creates 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.

The SDELD object has the following displayed “Properties” on page 15-179:

- 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

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

Note Although sdemrd does not enforce restrictions on the signs of Alpha or Sigma, each parameter is specified as a positive value.

Properties

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

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, 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.

Data Types: double

Correlation — Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes)
NBrowns-by-NBrowns identity matrix representing independent Gaussian processes (default) | positive semidefinite matrix | deterministic function

Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes), specified 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. If Correlation is not a symmetric positive semidefinite matrix, use nearcorr to create a positive semidefinite matrix for a correlation matrix.

A Correlation matrix represents a static condition.

As a deterministic function of time, Correlation allows you to specify a dynamic correlation structure.

Data Types: double

Simulation — User-defined simulation function or SDE simulation method
simulation by Euler approximation (simByEuler) (default) | function | SDE simulation method

User-defined simulation function or SDE simulation method, specified as a function or SDE simulation method.

Data Types: function_handle

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)

This property is read-only.

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 drift 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 displayed parameters for a drift object 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)`

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)

This property is read-only.

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 `diffusion`:

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

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 displayed parameters for a `diffusion` object 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)`

Data Types: `struct` | `double`

Object Functions

interpolate Brownian interpolation of stochastic differential equations (SDEs) for SDE, BM, GBM, CEV, CIR, HWV, Heston, SDEDDO, SDELD, or SDEMRD models

simulate Simulate multivariate stochastic differential equations (SDEs) for SDE, BM, GBM, CEV, CIR, HWV, Heston, SDEDDO, SDELD, SDEMRD, Merton, or Bates models

simByEuler Euler simulation of stochastic differential equations (SDEs) for SDE, BM, GBM, CEV, CIR, HWV, Heston, SDEDDO, SDELD, or SDEMRD models

simByMilstein Simulate BM, GBM, CEV, HWV, SDEDDO, SDELD, SDEMRD sample paths by Milstein approximation

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
```

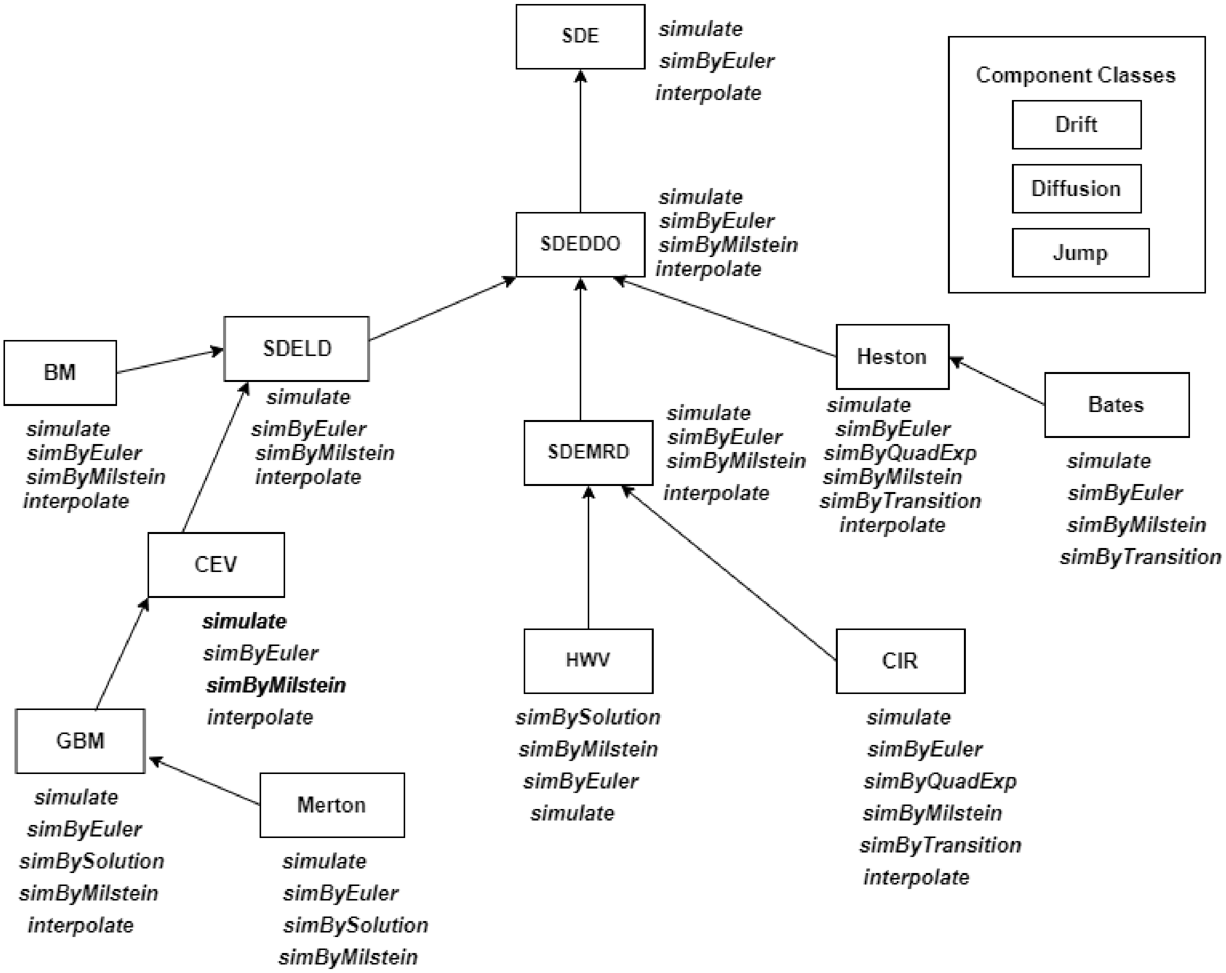
`sdemrd` objects display the familiar `Speed` and `Level` parameters instead of `A` and `B`.

More About

Instance Hierarchy

There are inheritance relationships among the SDE classes.

The following figure illustrates the inheritance relationships.



For more information, see “SDE Class Hierarchy” on page 14-5.

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.

Version History

Introduced in R2008a

References

[1] Aït-Sahalia, Yacine. “Testing Continuous-Time Models of the Spot Interest Rate.” *Review of Financial Studies*, vol. 9, no. 2, Apr. 1996, pp. 385–426.

[2] Aït-Sahalia, Yacine. “Transition Densities for Interest Rate and Other Nonlinear Diffusions.” *The Journal of Finance*, vol. 54, no. 4, Aug. 1999, pp. 1361–95.

[3] Glasserman, Paul. *Monte Carlo Methods in Financial Engineering*. Springer, 2004.

[4] Hull, John. *Options, Futures and Other Derivatives*. 7th ed, Prentice Hall, 2009.

[5] Johnson, Norman Lloyd, et al. *Continuous Univariate Distributions*. 2nd ed, Wiley, 1994.

[6] Shreve, Steven E. *Stochastic Calculus for Finance*. Springer, 2004.

See Also

`drift` | `diffusion` | `sdeddo` | `simByEuler` | `nearcorr`

Topics

“Creating Stochastic Differential Equations from Mean-Reverting Drift (SDEMRD) Models” on page 14-23
“Simulating Equity Prices” on page 14-28
“Simulating Interest Rates” on page 14-48
“Stratified Sampling” on page 14-57
“Price American Basket Options Using Standard Monte Carlo and Quasi-Monte Carlo Simulation” on page 14-70
“Base SDE Models” on page 14-14
“Drift and Diffusion Models” on page 14-16
“Linear Drift Models” on page 14-19
“Parametric Models” on page 14-21
“SDEs” on page 14-2
“SDE Models” on page 14-7
“SDE Class Hierarchy” on page 14-5
“Quasi-Monte Carlo Simulation” on page 14-62
“Performance Considerations” on page 14-64

bates

`Bates` stochastic volatility model

Description

The `bates` function creates a `bates` object, which represents a Bates model.

The Bates model is a bivariate composite model that derives from the `heston` object. The Bates model is composed of two coupled and dissimilar univariate models, each driven by a single Brownian motion source of risk and a single compound Poisson process representing the arrivals of important events over `NPeriods` consecutive observation periods. The Bates model approximates continuous-time Bates stochastic volatility processes.

The first univariate model is a `GBM` model with a stochastic volatility function and a stochastic jump process, and usually corresponds to a price process whose variance rate is governed by the second univariate model. The second model is a Cox-Ingersoll-Ross (`CIR`) square root diffusion model that describes the evolution of the variance rate of the coupled `GBM` price process.

Bates models are bivariate composite models. Each Bates model consists of two coupled univariate models:

- A geometric Brownian motion (`gbm`) model with a stochastic volatility function and jumps.

  $$dX_{1t} = B(t)X_{1t}dt + \sqrt{X_{2t}}X_{1t}dW_{1t} + Y(t)X_{1t}dN_t$$

  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 Bates price process.

Creation

Syntax

```
Bates = bates(Return,Speed,Level,Volatility,JumpFreq,JumpMean,JumpVol)
Bates = bates( ___ ,Name,Value)
```

Description

`Bates = bates(Return,Speed,Level,Volatility,JumpFreq,JumpMean,JumpVol)` create a `bates` object with the default options.

Since Bates models are bivariate models composed of coupled univariate models, all required inputs correspond to scalar parameters. Specify required inputs as one of two types:

- MATLAB array. Specify an array to indicate a static (non-time-varying) parametric specification. This array fully captures all implementation details, which are clearly associated with a parametric form.
- MATLAB function. Specify a function to provide indirect support for virtually any static, dynamic, linear, or nonlinear model. This parameter is supported by 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.

`Bates = bates(___,Name,Value)` sets “Properties” on page 15-189 using name-value pair arguments in addition to the input arguments in the preceding syntax. Enclose each property name in quotes.

The `bates` object has the following “Properties” on page 15-189:

- `StartTime` — Initial observation time
- `StartState` — Initial state at time `StartTime`
- `Correlation` — Access function for the `Correlation` input argument
- `Drift` — Composite drift-rate function
- `Diffusion` — Composite diffusion-rate function
- `Simulation` — A simulation function or method

Input Arguments

`Return` — Expected mean instantaneous rate of return of GBM price process

array | deterministic function of time | deterministic function of time and state

Expected mean instantaneous rate of return of the GBM price process, specified as an array, or a deterministic function of time.

If you specify `Return` as an array, it must be scalar.

If you specify `Return` as a deterministic function of time, you call `Return` with a real-valued scalar time `t` as its only input, it must return a scalar.

If you specify `Return` as a deterministic function of time and state, it must return a scalar when you call it with two inputs:

- A real-valued scalar observation time t
- A 2-by-1 bivariate state vector X_t

Data Types: `double` | `function_handle`

Speed — Mean-reversion speed of CIR stochastic variance process

array | deterministic function of time | deterministic function of time and state

Mean-reversion speed of the CIR stochastic variance process, specified as an array or deterministic function of time.

If you specify `Speed` as an array, it must be a scalar.

If you specify `Speed` as a deterministic function of time, you call `Speed` with a real-valued scalar time `t` as its only input, it must return a scalar.

If you specify `Speed` as a function of time and state, the function calculates the speed of mean reversion. This function must return a scalar of reversion rates when you call it with two inputs:

- A real-valued scalar observation time t
- A `2`-by-`1` bivariate state vector X_t

Note Although `bates` enforces no restrictions on `Speed`, the mean-reversion speed is nonnegative such that the underlying process reverts to some stable level.

Data Types: `double` | `function_handle`

`Level` — Reversion level or long-run average of CIR stochastic variance process

array | deterministic function of time | deterministic function of time and state

Reversion level or long-run average of the CIR stochastic variance process, specified as an array, or deterministic function of time.

If you specify `Level` as an array, it must be a scalar.

If you specify `Level` as a deterministic function of time, you call `Level` with a real-valued scalar time `t` as its only input, it must return a scalar.

If you specify `Level` as a deterministic function of time and state, it must return a scalar of reversion levels when you call it with two inputs:

- A real-valued scalar observation time t
- A `2`-by-`1` bivariate state vector X_t

Data Types: `double` | `function_handle`

`Volatility` — Instantaneous volatility CIR stochastic variance process

scalar | deterministic function of time | deterministic function of time and state

Instantaneous volatility of the CIR stochastic variance process (often called the *volatility of volatility* or *volatility of variance*), specified as a scalar, a deterministic function of time, or a deterministic function of time and state.

If you specify `Volatility` as a scalar, it represents the instantaneous volatility of the CIR stochastic variance model.

If you specify `Volatility` as a deterministic function of time, you call `Volatility` with a real-valued scalar time `t` as its only input, it must return a scalar.

If you specify `Volatility` as a deterministic function time and state, `Volatility` must return a scalar when you call it with two inputs:

- A real-valued scalar observation time t

- A 2-by-1 bivariate state vector X_t

Note Although `bates` enforces no restrictions on `Volatility`, the volatility is usually nonnegative.

Data Types: `double` | `function_handle`

`JumpFreq` — Instantaneous jump frequencies representing intensities of Poisson processes
array | deterministic function of time | deterministic function of time and state

Instantaneous jump frequencies representing the intensities (the mean number of jumps per unit time) of Poisson processes (N_t) that drive the jump simulation, specified as an array, a deterministic function of time, or a deterministic function of time and state.

If you specify `JumpFreq` as an array, it must be a scalar.

If you specify `JumpFreq` as a deterministic function of time, you call `JumpFreq` with a real-valued scalar time `t` as its only input, it must return a scalar.

If you specify `JumpFreq` as a function of time and state, `JumpFreq` must return a scalar when you call it with two inputs:

- A real-valued scalar observation time t
- A 2-by-1 bivariate state vector X_t

Data Types: `double` | `function_handle`

`JumpMean` — Instantaneous mean of random percentage jump sizes
array | deterministic function of time | deterministic function of time and state

Instantaneous mean of random percentage jump sizes J, where log(1+J) is normally distributed with mean (log(1+`JumpMean`) - 0.5 × `JumpVol`^{2}) and standard deviation `JumpVol`, specified as an array, a deterministic function of time, or a deterministic function of time and state.

If you specify `JumpMean` as an array, it must be a scalar.

If you specify `JumpMean` as a deterministic function of time, you call `JumpMean` with a real-valued scalar time `t` as its only input, it must return a scalar.

If you specify `JumpMean` as a function of time and state, `JumpMean` must return a scalar when you call it with two inputs:

- A real-valued scalar observation time t
- A 2-by-1 bivariate state vector X_t

Data Types: `double` | `function_handle`

`JumpVol` — Instantaneous standard deviation
array | deterministic function of time | deterministic function of time and state

Instantaneous standard deviation of log(1+J), specified as an array, a deterministic function of time, or a deterministic function of time and state.

If you specify `JumpVol` as an array, it must be a scalar.

If you specify JumpVol as a deterministic function of time, you call JumpVol with a real-valued scalar time t as its only input, it must return a scalar.

If you specify JumpVol as a function of time and state, JumpVol must return a scalar when you call it with two inputs:

- A real-valued scalar observation time *t*
- A 2-by-1 bivariate state vector X_t

Data Types: double | function_handle

Properties

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.

Data Types: double

StartState — Initial values of state variables
1 (default) | scalar | column vector | matrix

Initial values of state variables, specified as a scalar, column vector, or matrix.

If StartState is a scalar, bates applies the same initial value to all state variables on all trials.

If StartState is a bivariate column vector, bates applies a unique initial value to each state variable on all trials.

If StartState is a matrix, bates applies a unique initial value to each state variable on each trial.

Data Types: double

Correlation — Correlation between Gaussian random variates drawn to generate Brownian motion vector (Wiener processes)
2-by-2 identity matrix representing independent Gaussian processes (default) | positive semidefinite matrix | deterministic function

Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes), specified as a scalar, a 2-by-2 positive semidefinite matrix, or as a deterministic function C_t that accepts the current time *t* and returns an 2-by-2 positive semidefinite correlation matrix. If Correlation is not a symmetric positive semidefinite matrix, use nearcorr to create a positive semidefinite matrix for a correlation matrix.

A Correlation matrix represents a static condition.

If you specify Correlation as a deterministic function of time, Correlation allows you to specify a dynamic correlation structure.

Data Types: double

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)

This property is read-only.

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.

Use the `drift` function to create `drift` objects of the form

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

Here:

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

The displayed parameters for a `drift` object follow.

- `Rate` — Drift-rate function, $F(t,X_t)$
- `A` — Intercept term, $A(t,X_t)$, of $F(t,X_t)$
- `B` — 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`.

Specifying `AB` as MATLAB double arrays clearly associates them 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` objects 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)`

Data Types: `object`

`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)

This property is read-only.

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 for approximating continuous-time stochastic processes.

Use the `diffusion` function to create `diffusion` objects of the form

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

Here:

- `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 displayed parameters for a `diffusion` object are:

- `Rate` — Diffusion-rate function, $G(t,X_t)$
- `Alpha` — State vector exponent, which determines the format of $D(t,X_t)$ of $G(t,X_t)$
- `Sigma` — 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` objects 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)`

Data Types: `object`

`Simulation` — User-defined simulation function or SDE simulation method

simulation by Euler approximation (`simByEuler`) (default) | function | SDE simulation method

User-defined simulation function or SDE simulation method, specified as a function or SDE simulation method.

Data Types: `function_handle`

Object Functions

simByEuler	Simulate Bates sample paths by Euler approximation
simByQuadExp	Simulate Bates, Heston, and CIR sample paths by quadratic-exponential discretization scheme
simulate	Simulate multivariate stochastic differential equations (SDEs) for SDE, BM, GBM, CEV, CIR, HWV, Heston, SDEDDO, SDELD, SDEMRD, Merton, or Bates models
simByTransition	Simulate Bates sample paths with transition density
simByMilstein	Simulate Bates sample paths by Milstein approximation

Examples

Create bates Object

Bates models are bivariate composite models, composed of two coupled and dissimilar univariate models, each driven by a single Brownian motion source of risk and a single compound Poisson process representing the arrivals of important events over `NPeriods` consecutive observation periods. The simulation approximates continuous-time Bates stochastic volatility processes.

Create a `bates` object.

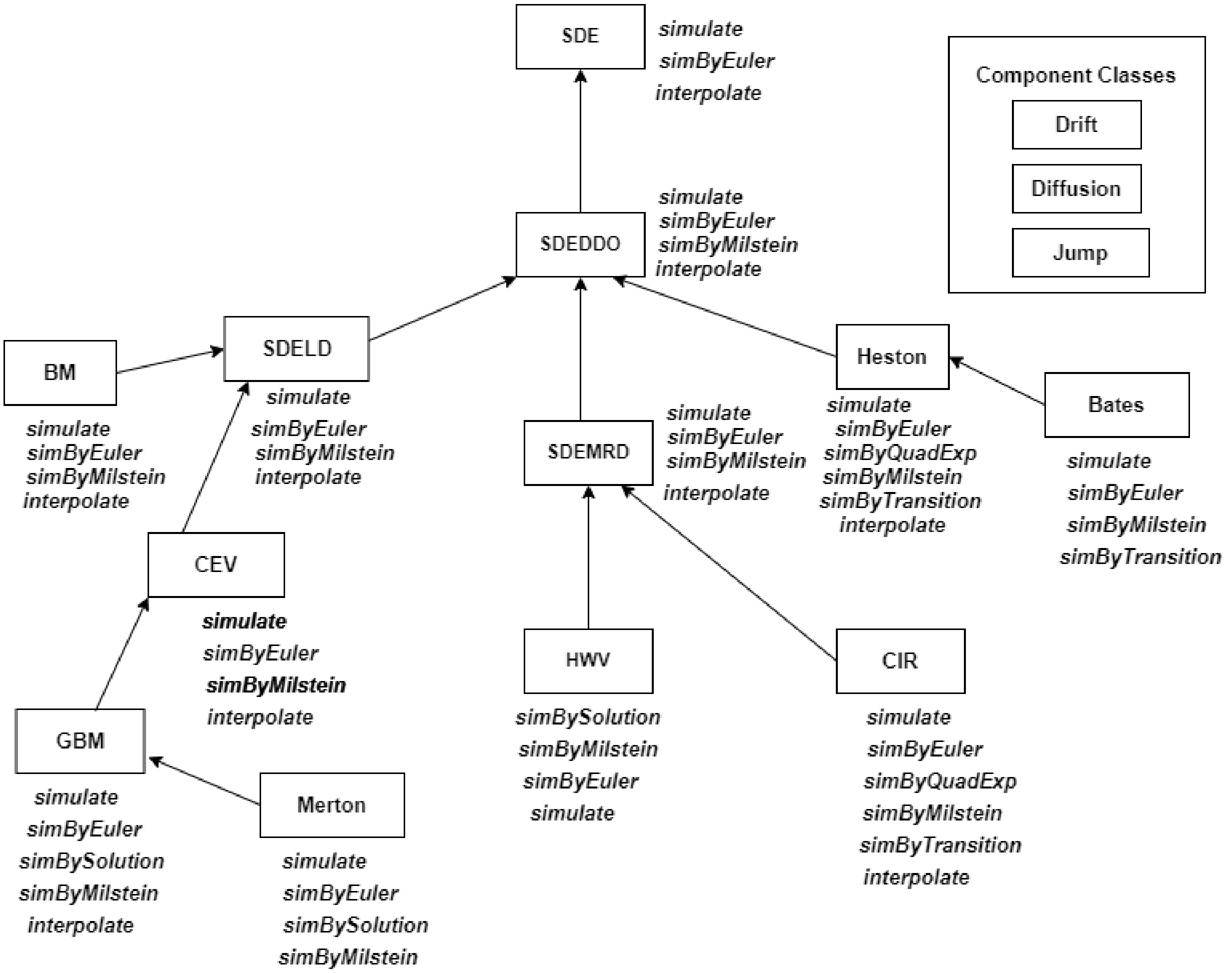
```
AssetPrice = 80;
            Return = 0.03;
            JumpMean = 0.02;
            JumpVol = 0.08;
            JumpFreq = 0.1;

            V0 = 0.04;
            Level = 0.05;
            Speed = 1.0;
            Volatility = 0.2;
            Rho = -0.7;
            StartState = [AssetPrice;V0];
            Correlation = [1 Rho;Rho 1];

batesObj = bates(Return, Speed, Level, Volatility,...
                JumpFreq, JumpMean, JumpVol,'startstate',StartState,...
                'correlation',Correlation)
```

```
batesObj = 
   Class BATES: Bates Bivariate Stochastic Volatility
   --------------------------------------------------
     Dimensions: State = 2, Brownian = 2
   --------------------------------------------------
      StartTime: 0
     StartState: 2x1 double array
    Correlation: 2x2 double array
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
         Return: 0.03
          Speed: 1
          Level: 0.05
     Volatility: 0.2
       JumpFreq: 0.1
       JumpMean: 0.02
        JumpVol: 0.08
```

More About

Instance Hierarchy

There are inheritance relationships among the SDE classes, as follows.



For more information, see “SDE Class Hierarchy” on page 14-5.

Algorithms

The Bates model (Bates 1996) is an extension of the Heston model and adds not only stochastic volatility, but also the jump diffusion parameters as in Merton (1976) were also added to model sudden asset price movements.

Under the risk-neutral measure the model is expressed as follows

$$dS_t = (\gamma - q - \lambda_p \mu_j)S_t dt + \sqrt{\upsilon_t} S_t dW_t + J S_t dP_t$$

$$d\upsilon_t = \kappa(\theta - \upsilon_t)dt + \sigma_\upsilon \sqrt{\upsilon_t} dW_t^\upsilon$$

$$\mathrm{E}\left[dW_t dW_t^\upsilon\right] = pdt$$

$$\mathrm{prob}(dP_t = 1) = \lambda_p dt$$

Here:

$^\gamma$ is the continuous risk-free rate.

q is the continuous dividend yield.

J is the random percentage jump size conditional on the jump occurring, where

$$\ln(1 + J) \sim N(\ln(1+u_j) - \frac{\delta^2}{2}, \delta^2$$

(1+J) has a lognormal distribution:

$$\frac{1}{(1 + J)\delta\sqrt{2\pi}}\exp\left\{\frac{-\left[\ln(1 + J) - (\ln(1+\mu_j) - \frac{\delta^2}{2}\right]^2}{2\delta^2}\right\}$$

Here:

μ_j is the mean of J($\mu_j > -1$).

ƛ$_p$ is the annual frequency (intensity) of the Poisson process P_t (ƛ$_p \geq 0$).

υ is the initial variance of the underlying asset ($\upsilon_0 > 0$).

θ is the long-term variance level (θ > 0).

κ is the mean reversion speed for the variance (κ > 0).

σ_υ is the volatility of volatility ($\sigma_\upsilon > 0$).

p is the correlation between the Weiner processes W_t and W_t^υ ($-1 \leq p \leq 1$).

The "Feller condition" ensures positive variance: ($2\kappa\theta > \sigma_\upsilon^2$).

The stochastic volatility along with the jump help better model the asymmetric leptokurtic features, the volatility smile, and the large random fluctuations such as crashes and rallies.

Version History

Introduced in R2020a

R2023a: Added `simByMilstein` method

Behavior changed in R2023a

Use the `simByMilstein` method to approximate a numerical solution of a stochastic differential equation.

References

[1] Aït-Sahalia, Yacine. “Testing Continuous-Time Models of the Spot Interest Rate.” *Review of Financial Studies* 9, no. 2 (Apr. 1996): 385–426.

[2] Aït-Sahalia, Yacine. “Transition Densities for Interest Rate and Other Nonlinear Diffusions.” *The Journal of Finance* 54, no. 4 (Aug. 1999): 1361–95.

[3] Glasserman, Paul. *Monte Carlo Methods in Financial Engineering*. New York: Springer-Verlag, 2004.

[4] Hull, John C. *Options, Futures and Other Derivatives*. 7th ed, Prentice Hall, 2009.

[5] Johnson, Norman Lloyd, Samuel Kotz, and Narayanaswamy Balakrishnan. *Continuous Univariate Distributions*. 2nd ed. Wiley Series in Probability and Mathematical Statistics. New York: Wiley, 1995.

[6] Shreve, Steven E. *Stochastic Calculus for Finance*. New York: Springer-Verlag, 2004.

See Also

`simByEuler` | `merton` | `simulate`

Topics

"Simulating Equity Prices" on page 14-28
"Simulating Interest Rates" on page 14-48
"Stratified Sampling" on page 14-57
"Price American Basket Options Using Standard Monte Carlo and Quasi-Monte Carlo Simulation" on page 14-70
"Base SDE Models" on page 14-14
"Drift and Diffusion Models" on page 14-16
"Linear Drift Models" on page 14-19
"Parametric Models" on page 14-21
"SDEs" on page 14-2
"SDE Models" on page 14-7
"SDE Class Hierarchy" on page 14-5
"Quasi-Monte Carlo Simulation" on page 14-62
"Performance Considerations" on page 14-64

merton

Merton jump diffusion model

Description

The merton function creates a merton object, which derives from the gbm object.

The merton model, based on the Merton76 model, allows you to simulate sample paths of NVars state variables driven by NBrowns Brownian motion sources of risk and NJumps compound Poisson processes representing the arrivals of important events over NPeriods consecutive observation periods. The simulation approximates continuous-time merton stochastic processes.

You can simulate any vector-valued merton process of the form

$$dX_t = B(t, X_t)X_t dt + D(t, X_t)V(t, x_t)dW_t + Y(t, X_t, N_t)X_t dN_t$$

Here:

- X_t is an NVars-by-1 state vector of process variables.
- $B(t,X_t)$ is an NVars-by-NVars matrix of generalized expected instantaneous rates of return.
- $D(t,X_t)$ is an NVars-by-NVars diagonal matrix in which each element along the main diagonal is the corresponding element of the state vector.
- $V(t,X_t)$ is an NVars-by-NVars matrix of instantaneous volatility rates.
- dW_t is an NBrowns-by-1 Brownian motion vector.
- $Y(t,X_t,N_t)$ is an NVars-by-NJumps matrix-valued jump size function.
- dN_t is an NJumps-by-1 counting process vector.

Creation

Syntax

```
Merton = merton(Return,Sigma,JumpFreq,JumpMean,JumpVol)
Merton = merton( ___ ,Name,Value)
```

Description

Merton = merton(Return,Sigma,JumpFreq,JumpMean,JumpVol) creates a default merton object. Specify required inputs as one of two types:

- MATLAB array. Specify an array to indicate a static (non-time-varying) parametric specification. This array fully captures all implementation details, which are clearly associated with a parametric form.
- MATLAB function. Specify a function to provide indirect support for virtually any static, dynamic, linear, or nonlinear model. This parameter is supported by 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.

`Merton = merton( ___ ,Name,Value)` sets "Properties" on page 15-199 using name-value pair arguments in addition to the input arguments in the preceding syntax. Enclose each property name in quotes.

The `merton` object has the following "Properties" on page 15-199:

- `StartTime` — Initial observation time
- `StartState` — Initial state at time `StartTime`
- `Correlation` — Access function for the `Correlation` input argument
- `Drift` — Composite drift-rate function
- `Diffusion` — Composite diffusion-rate function
- `Simulation` — A simulation function or method

Input Arguments

`Return` — Expected mean instantaneous rates of asset return

array | deterministic function of time | deterministic function of time and state

Expected mean instantaneous rates of asset return, denoted as $B(t,X_t)$, specified as an array, a deterministic function of time, or a deterministic function of time and state.

If you specify `Return` as an array, it must be an `NVars`-by-`NVars` matrix representing the expected (mean) instantaneous rate of return.

If you specify Return as a deterministic function of time, when you call `Return` with a real-valued scalar time `t` as its only input, it must return an `NVars`-by-`NVars` matrix.

If you specify `Return` as a deterministic function of time and state, it must return an `NVars`-by-`NVars` matrix when you call it with two inputs:

- A real-valued scalar observation time t
- An `NVars`-by-`1` state vector X_t

Data Types: `double` | `function_handle`

`Sigma` — Instantaneous volatility rates

array | deterministic function of time | deterministic function of time and state

Instantaneous volatility rates, denoted as $V(t,X_t)$, specified as an array, a deterministic function of time, or a deterministic function of time and state.

If you specify `Sigma` as an array, it must be an `NVars`-by-`NBrowns` matrix of instantaneous volatility rates or 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.

If you specify `Sigma` as a deterministic function of time, when you call `Sigma` with a real-valued scalar time `t` as its only input, it must return an `NVars`-by-`NBrowns` matrix.

If you specify Sigma as a deterministic function of time and state, it must return an NVars-by-NBrowns matrix when you call it with two inputs:

- A real-valued scalar observation time t
- An NVars-by-1 state vector X_t

Note Although merton enforces no restrictions for Sigma, volatilities are usually nonnegative.

Data Types: double | function_handle

JumpFreq — Instantaneous jump frequencies representing intensities of Poisson processes
array | deterministic function of time | deterministic function of time and state

Instantaneous jump frequencies representing the intensities (the mean number of jumps per unit time) of the Poisson processes N_t that drive the jump simulation, specified as an array, a deterministic function of time, or a deterministic function of time and state.

If you specify JumpFreq as an array, it must be an NJumps-by-1 vector.

If you specify JumpFreq as a deterministic function of time, when you call JumpFreq with a real-valued scalar time t as its only input, JumpFreq must produce an NJumps-by-1 vector.

If you specify JumpFreq as a deterministic function of time and state, it must return an NVars-by-NBrowns matrix when you call it with two inputs:

- A real-valued scalar observation time t
- An NVars-by-1 state vector X_t

Data Types: double | function_handle

JumpMean — Instantaneous mean of random percentage jump sizes
array | deterministic function of time | deterministic function of time and state

Instantaneous mean of random percentage jump sizes J, where log(1+J) is normally distributed with mean (log(1+JumpMean) - 0.5 × JumpVol^2) and standard deviation JumpVol, specified as an array, a deterministic function of time, or a deterministic function of time and state.

If you specify JumpMean as an array, it must be an NVars-by-NJumps matrix.

If you specify JumpMean as a deterministic function of time, when you cal JumpMean with a real-valued scalar time t as its only input, it must return an NVars-by-NJumps matrix.

If you specify JumpMean as a deterministic function of time and state, it must return an NVars-by-NJumps matrix when you call it with two inputs:

- A real-valued scalar observation time t
- An NVars-by-1 state vector X_t

Data Types: double | function_handle

JumpVol — Instantaneous standard deviation
array | deterministic function of time | deterministic function of time and state

Instantaneous standard deviation of log(1+*J*), specified as an array, a deterministic function of time, or a deterministic function of time and state.

If you specify JumpVol as an array, it must be an NVars-by-NJumps matrix.

If you specify JumpVol as a deterministic function of time, when you call JumpVol with a real-valued scalar time t as its only input, it must return an NVars-by-NJumps matrix.

If you specify JumpVol as a deterministic function of time and state, it must return an NVars-by-NJumps matrix when you call it with two inputs:

- A real-valued scalar observation time *t*
- An NVars-by-1 state vector X_t

Data Types: double | function_handle

Properties

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.

Data Types: double

StartState — Initial values of state variables

1 (default) | scalar | column vector | matrix

Initial values of state variables, specified as a scalar, column vector, or matrix.

If StartState is a scalar, merton applies the same initial value to all state variables on all trials.

If StartState is a column vector, merton applies a unique initial value to each state variable on all trials.

If StartState is a matrix, merton applies a unique initial value to each state variable on each trial.

Data Types: double

Correlation — Correlation between Gaussian random variates drawn to generate Brownian motion vector (Wiener processes)

NBrowns-by-NBrowns identity matrix representing independent Gaussian processes (default) | positive semidefinite matrix | deterministic function

Correlation between Gaussian random variates drawn to generate the Brownian motion vector (Wiener processes), specified 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. If Correlation is not a symmetric positive semidefinite matrix, use nearcorr to create a positive semidefinite matrix for a correlation matrix.

A Correlation matrix represents a static condition.

If you specify Correlation as a deterministic function of time, you can specify a dynamic correlation structure.

Data Types: double

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)

This property is read-only.

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.

Use the drift function to create drift objects of the form

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

Here:

- 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 displayed parameters for a drift object are:

- Rate — Drift-rate function, $F(t,X_t)$
- A — Intercept term, $A(t,X_t)$, of $F(t,X_t)$
- B — 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.

Specifying AB as MATLAB double arrays clearly associates them 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 objects 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)`

Data Types: object

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)

This property is read-only.

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 for approximating continuous-time stochastic processes.

Use the diffusion function to create diffusion objects of the form

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

Here:

- 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 displayed parameters for a diffusion object are:

- Rate — Diffusion-rate function, $G(t,X_t)$
- Alpha — State vector exponent, which determines the format of $D(t,X_t)$ of $G(t,X_t)$
- Sigma — 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 objects 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)`

Data Types: object

Simulation — User-defined simulation function or SDE simulation method

simulation by Euler approximation (simByEuler) (default) | function | SDE simulation method

User-defined simulation function or SDE simulation method, specified as a function or SDE simulation method.

Data Types: function_handle

Object Functions

simByEuler	Simulate Merton jump diffusion sample paths by Euler approximation
simBySolution	Simulate approximate solution of diagonal-drift Merton jump diffusion process
simByMilstein	Simulate Merton sample paths by Milstein approximation
simulate	Simulate multivariate stochastic differential equations (SDEs) for SDE, BM, GBM, CEV, CIR, HWV, Heston, SDEDDO, SDELD, SDEMRD, Merton, or Bates models

Examples

Create merton Object

Merton jump diffusion models allow you to simulate sample paths of NVARS state variables driven by NBROWNS Brownian motion sources of risk and NJumps compound Poisson processes representing the arrivals of important events over NPeriods consecutive observation periods. The simulation approximates continuous-time merton stochastic processes.

Create a merton object.

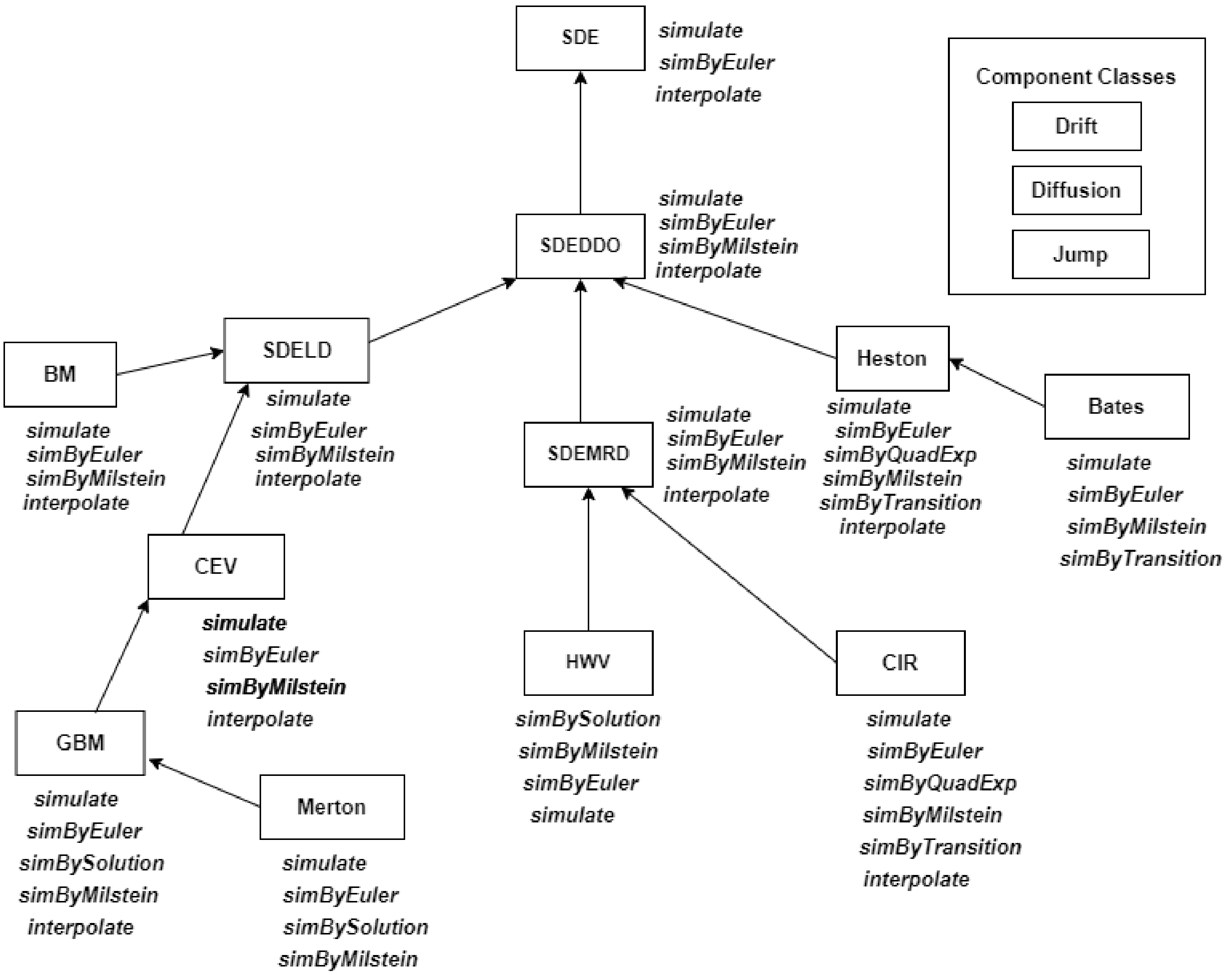
```
AssetPrice = 80;
            Return = 0.03;
            Sigma = 0.16;
            JumpMean = 0.02;
            JumpVol = 0.08;
            JumpFreq = 2;

            mertonObj = merton(Return,Sigma,JumpFreq,JumpMean,JumpVol,...
                'startstat',AssetPrice)
```

```
mertonObj = 
   Class MERTON: Merton Jump Diffusion
   ----------------------------------------
     Dimensions: State = 1, Brownian = 1
   ----------------------------------------
      StartTime: 0
     StartState: 80
    Correlation: 1
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
          Sigma: 0.16
         Return: 0.03
       JumpFreq: 2
       JumpMean: 0.02
        JumpVol: 0.08
```

More About

Instance Hierarchy

There are inheritance relationships among the SDE classes, as follows.



For more information, see “SDE Class Hierarchy” on page 14-5.

Algorithms

The Merton jump diffusion model (Merton 1976) is an extension of the Black-Scholes model, and models sudden asset price movements (both up and down) by adding the jump diffusion parameters with the Poisson process P_t.

Under the risk-neutral measure the model is expressed as follows

$$dS_t = (\gamma - q - \lambda_p \mu_j) S_t dt + \sigma_M S_t dW_t + J S_t dP_t$$
$$\text{prob}(dP_t = 1) = \lambda_p dt$$

Here:

γ is the continuous risk-free rate.

q is the continuous dividend yield.

J is the random percentage jump size conditional on the jump occurring, where

$$\ln(1+J) \sim N(\ln(1+u_j) - \frac{\delta^2}{2}, \delta^2$$

(1+J) has a lognormal distribution:

$$\frac{1}{(1+J)\delta\sqrt{2\pi}}\exp\left\{\frac{-\left[\ln(1+J) - (\ln(1+\mu_j) - \frac{\delta^2}{2}\right]^2}{2\delta^2}\right\}$$

Here:

μ_j is the mean of $J(\mu_j > -1)$.

λ_p is the annual frequency (intensity) of the Poisson process P_t ($\lambda_p \geq 0$).

σ_M is the volatility of the asset price ($\sigma_M > 0$).

Under this formulation, extreme events are explicitly included in the stochastic differential equation as randomly occurring discontinuous jumps in the diffusion trajectory. Therefore, the disparity between observed tail behavior of log returns and that of Brownian motion is mitigated by the inclusion of a jump mechanism.

Version History

Introduced in R2020a

R2023a: Added `simByMilstein` method

Behavior changed in R2023a

Use the `simByMilstein` method to approximate a numerical solution of a stochastic differential equation.

References

[1] Aït-Sahalia, Yacine. “Testing Continuous-Time Models of the Spot Interest Rate.” *Review of Financial Studies* 9, no. 2 (Apr. 1996): 385–426.

[2] Aït-Sahalia, Yacine. “Transition Densities for Interest Rate and Other Nonlinear Diffusions.” *The Journal of Finance* 54, no. 4 (Aug. 1999): 1361–95.

[3] Glasserman, Paul. *Monte Carlo Methods in Financial Engineering*. New York: Springer-Verlag, 2004.

[4] Hull, John C. *Options, Futures and Other Derivatives*. 7th ed, Prentice Hall, 2009.

[5] Johnson, Norman Lloyd, Samuel Kotz, and Narayanaswamy Balakrishnan. *Continuous Univariate Distributions*. 2nd ed. Wiley Series in Probability and Mathematical Statistics. New York: Wiley, 1995.

[6] Shreve, Steven E. *Stochastic Calculus for Finance*. New York: Springer-Verlag, 2004.

See Also

`bates` | `simByEuler` | `simBySolution` | `simulate`

Topics

“Simulating Equity Prices” on page 14-28
“Simulating Interest Rates” on page 14-48
“Stratified Sampling” on page 14-57
“Price American Basket Options Using Standard Monte Carlo and Quasi-Monte Carlo Simulation” on page 14-70
“Base SDE Models” on page 14-14
“Drift and Diffusion Models” on page 14-16
“Linear Drift Models” on page 14-19
“Parametric Models” on page 14-21
“SDEs” on page 14-2
“SDE Models” on page 14-7
“SDE Class Hierarchy” on page 14-5
“Quasi-Monte Carlo Simulation” on page 14-62
“Performance Considerations” on page 14-64

backtestStrategy

Create `backtestStrategy` object to define portfolio allocation strategy

Description

Create a `backtestStrategy` object which defines a portfolio allocation strategy.

Use this workflow to develop and run a backtest:

1. Define the strategy logic using a `backtestStrategy` object to specify how the strategy rebalances a portfolio of assets.
2. Use `backtestEngine` to create a `backtestEngine` object that specifies the parameters of the backtest.
3. Use `runBacktest` to run the backtest against historical asset price data and, optionally, trading signal data.
4. Use `equityCurve` to plot the equity curves of each strategy.
5. Use `summary` to summarize the backtest results in a table format.

For more detailed information on this workflow, see “Backtest Investment Strategies Using Financial Toolbox™” on page 4-231.

Creation

Syntax

```
strategy = backtestStrategy(name,rebalanceFcn)
strategy = backtestStrategy( ___ ,Name,Value)
```

Description

`strategy = backtestStrategy(name,rebalanceFcn)` creates a `backtestStrategy` object.

`strategy = backtestStrategy( ___ ,Name,Value)` sets properties on page 15-213 using name-value pair arguments and any of the arguments in the previous syntax. You can specify multiple name-value pair arguments. For example, `strat = backtestStrategy('MyStrategy',@myRebalFcn,'TransactionCost',0.005,'LookbackWindow',20)`.

Input Arguments

name — Strategy name
string

Strategy name, specified as a string.

Data Types: `string`

rebalanceFcn — Rebalance function

function handle

Rebalance function, specified as a function handle which computes new portfolio weights during the backtest. The rebalanceFcn argument implements the core logic of the trading strategy.

The signature of the rebalanceFcn depends on whether UserData and EngineDataList name-value arguments are specified for the backtestStrategy function. For an example of rebalanceFcn where UserData is specified, see “Backtest with Brinson Attribution to Evaluate Portfolio Performance” on page 4-311.

- **If UserData and EngineDataList are Empty**

  If the UserData and EngineDataList name-value arguments are empty, then the rebalanceFcn must have one of the following signatures:

  - `new_weights = rebalanceFcn(weights,assetPrices)`
  - `new_weights = rebalanceFcn(weights,assetPrices,signalData)`

  If you do *not* specify UserData, the rebalance function returns a single output argument, new_weights, which is a vector of asset weights specified as decimal percentages.

  - If the new_weights sum to 1, then the portfolio is fully invested.
  - If the new_weights sum to less than 1, then the portfolio has the remainder in cash, earning the RiskFreeRate specified in the backtestEngine object.
  - If the new_weights sum to more than 1, then there is a negative cash position (margin) and the cash borrowed accrues interest at the cash borrowing rate specified in the CashBorrowRate property of the backtestEngine object.

- **If UserData is Empty and EngineDataList is Specified**

  If the UserData name-value argument is empty and the EngineDataList name-value argument is specified, then the rebalanceFcn must have one of the following signatures:

  - `new_weights = rebalanceFcn(engineData,assetPrices)`
  - `new_weights = rebalanceFcn(engineData,assetPrices,signalData)`

- **If UserData is Specified and EngineDataList is Empty**

  If you use the name-value argument for UserData to specify a strategy-specific userData struct but do not specify the name-value argument for EngineDataList, the required syntax for rebalanceFcn must have one of the following signatures:

  - `[new_weights,userData] = rebalanceFcn(weights,assetPrices,userData)`
  - `[new_weights,userData] = rebalanceFcn(weights,assetPrices,signalData,userData)`

  If UserData is specified, then in addition to an output for new_weights, there is also an output for userData returned as a struct.

- **If Both UserData and EngineDataList are Specified**

If you use the name-value argument for UserData to specify a strategy-specific userData struct and you use the name-value argument EngineDataList to specify a backtest engine state, the required syntax for rebalanceFcn must have one of the following signatures:

- [new_weights,userData] = rebalanceFcn(engineData,assetPrices,userData)
- [new_weights,userData] = rebalanceFcn(engineData,assetPrices,signalData,userData)

If UserData is specified, then in addition to an output for new_weights, there is also an output for userData returned as a struct.

The rebalanceFcn function is called by the backtestEngine object each time the strategy must be rebalanced as specified in the RebalanceFrequency name-value argument. The backtestEngine object calls the rebalanceFcn function with the following arguments:

- weights — The current portfolio weights before rebalancing, specified as decimal percentages.
- assetPrices — A timetable containing a rolling window of adjusted asset prices.
- signalData — (Optional) A timetable containing a rolling window of signal data.

  If you provide signal data to the backtestEngine object, then the engine object passes it to the strategy rebalance function using the three input argument syntax. If do not provide signal data the backtestEngine object, then the engine object calls the rebalance function with the two input argument syntax.
- userData — (Optional) A struct to contain strategy-specific user data.

  If the UserData property is set, the userData struct is passed into rebalanceFcn.
- engineData — (Optional) A struct containing backtest state data.

  If the EngineDataList property is set, the engineData struct is passed into rebalanceFcn.

For more information on developing a rebalanceFcn function handle, see “Backtest Investment Strategies Using Financial Toolbox™” on page 4-231.

Data Types: function_handle

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: strat = backtestStrategy('MyStrategy',@myRebalFcn,'TransactionCost',0.005,'LookbackWindow',20)

RebalanceFrequency — Rebalance frequency during backtest

1 (default) | integer | duration object | calendarDuration object | vector of datetime objects

Rebalance frequency during the backtest, specified as the comma-separated pair consisting of 'RebalanceFrequency' and a scalar integer, duration orcalendarDuration object, or a vector of datetime objects.

The RebalanceFrequency specifies the schedule of dates where the strategy will rebalance its portfolio. The default is 1, meaning the strategy rebalances with each time step.

For more information, see "Defining Schedules for Backtest Strategies" on page 15-226.

Data Types: double | object | datetime

TransactionCosts — Transaction costs for trades

0 (not computed) (default) | numeric | vector | function handle

Transaction costs for trades, specified as the comma-separated pair consisting of 'TransactionCosts' and a scalar numeric, vector, or function handle. You can specify transaction costs in three ways:

- rate — A scalar decimal percentage charge to both purchases and sales of assets. For example ,if you set TransactionCosts to 0.001, then each transaction (buys and sells) would pay 0.1% in transaction fees.
- [buyRate, sellRate] — A 1-by-2 vector of decimal percentage rates that specifies separate rates for buying and selling of assets.
- computeTransactionCostsFcn — A function handle to compute customized transaction fees. If you specify a function handle, the backtestEngine object calls the TransactionCosts function to compute the fees for each rebalance. The user-defined computeTransactionCostsFcn function handle has different signatures depending on whether the optional name-value arguments UserData and EngineDataList are specified.

  - **If UserData is Empty and EngineDataList is Empty**

    When the optional name-value arguments UserData and EngineDataList are not specified, the computeTransactionCostsFcn function handle has the following signature:

    ```
    [buyCosts,sellCosts] = computeTransactionCostsFcn(deltaPositions)
    ```

    The user-defined computeTransactionCostsFcn function handle takes a single input argument, deltaPositions, which is a vector of changes in asset positions for all assets (in currency units) as a result of a rebalance. Positive elements in the deltaPositions vector indicate purchases while negative entries represent sales. The user-defined function handle must return two output arguments buyCosts and sellCosts, which contain the total costs (in currency) for the entire rebalance for each type of transaction.
  - **If UserData is Empty and EngineDataList is Specified**

    When the optional name-value argument UserData is not specified, but the optional name-value argument EngineDataList is specified, the computeTransactionCostsFcn function handle has the following signatures:

    ```
    [buyCosts,sellCosts] = computeTransactionCostsFcn(deltaPositions,engineData)
    ```
  - **If UserData is Specified and EngineDataList is Empty**

    When the optional name-value argument UserData is specified, but the optional name-value argument EngineDataList is not specified, the computeTransactionCostsFcn function handle has the following signatures:

    ```
    [buyCosts,sellCosts,userData] = computeTransactionCostsFcn(deltaPositions,userData)
    ```
  - **If Both UserData and EngineDataList are Specified**

If you use the optional name-value argument for UserData to specify a strategy-specific userData struct and the optional name-value argument EngineDataList to specify required backtest engine state data, the required syntax for the computeTransactionCostsFcn function handle must have one of the following signatures:

```
[buyCosts,sellCosts,userData] = computeTransactionCostsFcn(deltaPositions,engineData,userDa
```

If you specify the optional name-value argument UserData, then the strategy userData struct is passed to the computeTransactionCostsFcn cost function as the final input argument. The computeTransactionCostsFcn function handle can read or write any information to the userData struct. The updated userData struct is returned as the third output argument.

Data Types: double | function_handle

LookbackWindow — Lookback window

[0 Inf] (default) | 1-by-2 vector using integers | duration object | calendarDuration object

Lookback window, specified as the comma-separated pair consisting of 'LookbackWindow' and a 1-by-2 vector of integers, a duration orcalendarDurationobject.

When using a 1-by-2 vector with integers that defines the minimum and maximum size of the rolling window of data (asset prices and signal data) that you provide to the rebalanceFcn argument, you specify these limits in terms of the number of time steps. When specified as integers, the lookback window is defined in terms of rows of data from the asset (pricesTT) and signal (signalTT) timetables used in the backtest. The lookback minimum sets the minimum number of rows of asset price data that must be available to the rebalance function before a strategy rebalance can occur. The lookback maximum sets the maximum size for the rolling window of price data that is passed to the rebalance function.

For example, if the backtestEngine object is provided with daily price data, then LookbackWindow specifies the size bounds of the rolling window in days. The default is [0 Inf], meaning that all available past data is given to the rebalance function. If you specify a non-zero minimum, then the software does not call rebalanceFcn until enough time steps process to meet the minimum size.

If you specify LookbackWindow as a single scalar value, then the value is both the minimum and maximum of the LookbackWindow (that is, a fixed-sized window).

If using a duration or calendarDuration object, the lookback window minimum and maximum are defined in terms of timespans relative to the time at a rebalance. For example if the lookback minimum was set to five days (that is, days(5)), the rebalance will only occur if the backtest start time is at least five days prior to the rebalance time. Similarly, if the lookback maximum was set to six months (that is, calmonths(6)), the lookback window would contain only data that occurred at six months prior to the rebalance time or later.

Note Alternatively, the LookbackWindow can be set to a single scalar value indicating that the rolling window should be exactly that size (either in terms of rows or a time duration). The minimum and maximum size will both be set to the provided value.

Data Types: double | object

InitialWeights — Initial portfolio weights

[] (default) | vector

Initial portfolio weights, specified as the comma-separated pair consisting of `'InitialWeights'` and a vector. The `InitialWeights` vector sets the portfolio weights before the `backtestEngine` object begins the backtest. The size of the initial weights vector must match the number of assets used in the backtest.

Alternatively, you can set the `InitialWeights` name-value pair argument to empty (`[]`) to indicate the strategy will begin with no investments and in a 100% cash position. The default for `InitialWeights` is empty (`[ ]`).

Data Types: `double`

`ManagementFee` — Management fee charged as an annualized percent of the total portfolio value
`0` (default) | decimal

Management fee charged as an annualized percent of the total portfolio value, specified as the comma-separated pair consisting of `'ManagementFee'` and a numeric scalar. The management fee is the annualized fee rate paid to cover the costs of managing a strategy's portfolio. The management fee is charged based on the strategy portfolio balance at the start of each date specified by the `ManagementFeeSchedule`. The default is `0`, meaning no management fee is paid.

For more information, see “Management Fees” on page 15-227.

Data Types: `double`

`ManagementFeeSchedule` — Management fee schedule
`calyears(1)` (default) | integer | vector of datetimes | `duration` object | `calendarDuration` object

Management fee schedule, specified as the comma-separated pair consisting of `'ManagementFeeSchedule'` and a numeric scalar, a `duration` or `calendarDuration` object, or alternatively, as a vector of datetimes.

The `ManagementFeeSchedule` specifies the schedule of dates where the management fee is charged (if a `ManagementFee` is defined). The default is `calyears(1)`, meaning the management fee is charged annually.

For more information, see “Defining Schedules for Backtest Strategies” on page 15-226.

Data Types: `double` | `datetime` | `object`

`PerformanceFee` — Performance fee charged as an absolute percent of the total portfolio growth
`0` (default) | decimal

Performance fee charged as a percent of the total portfolio growth, specified as the comma-separated pair consisting of `'PerformanceFee'` and a numeric scalar. The performance fee, sometimes called an incentive fee, is a fee paid to reward a fund manager based on performance of the fund. The fee is paid periodically based on the schedule specified by the `PerformanceFeeSchedule`. The default is `0`, meaning no performance fee is paid.

For more information, see “Performance Fees” on page 15-229.

Data Types: `double`

PerformanceHurdle — Annualized hurdle rate or column in assets or signals to act as hurdle asset

0 (default) | decimal | string

Annualized hurdle rate or column in assetPrices or signalData to act as hurdle asset, specified as the comma-separated pair consisting of 'PerformanceHurdle' and a numeric scalar or a string. The PerformanceHurdle sets a minimum level of performance that a strategy must achieve before the performance fee is paid.

- If specified as a numeric scalar, the hurdle represents an annualized growth rate, for example 0.05 indicates a 5% growth rate. In this case, the performance fee is only paid if, on a performance fee date, the strategy portfolio value is greater than the high-water mark and the portfolio has produced a return greater than 5% annualized since the start of the backtest. The fee is paid only on the growth in excess of both the hurdle rate and the previous high-water mark.
- If specified as a string, the hurdle must match a column in either the assetPrices or signalData timetables of the backtest and it indicates a hurdle asset. In this case, the performance fee is paid only if the portfolio value is greater than the high-water mark and the portfolio has produced a return greater than the hurdle asset return since the start of the backtest. The performance fee is charged only on the portfolio growth in excess of both the hurdle asset growth and the previous high-water mark.

For more information, see “Performance Hurdle” on page 15-230.

Data Types: double | string

PerformanceFeeSchedule — Performance fee schedule

calyears(1) (default) | integer | vector of datetimes | duration object | calendarDuration object

Performance fee schedule, specified as the comma-separated pair consisting of 'PerformanceFeeSchedule' and a numeric scalar, a duration or calendarDuration object, or alternatively as a vector of datetimes.

PerformanceFeeSchedule specifies the schedule of dates where the performance fee is charged (if PerformanceFee is defined). The default is calyears(1), meaning the performance fee is charged annually.

For more information, see “Defining Schedules for Backtest Strategies” on page 15-226.

Data Types: double | datetime | object

UserData — Strategy-specific user data for use in rebalanceFcn

[] (default) | struct

Strategy-specific user data for use in rebalanceFcn, specified as a struct. When UserData is set to a struct, the backtestEngine provides all user-defined rebalanceFcn function handle functions with the userData struct as an additional input argument. The rebalanceFcn function handle functions can read and write to the userData struct. The rebalanceFcn function handle functions return the updated userData struct as an additional output argument.

For an example of using UserData, see “Use UserData Property to Create Stop Loss Trading Strategy” on page 15-219 and “Backtest with Brinson Attribution to Evaluate Portfolio Performance” on page 4-311.

Data Types: struct

EngineDataList — Specify set of optional backtest state data a strategy needs in rebalanceFcn

[ ] (default) | string | string array

Specify set of optional backtest state data a strategy needs in the rebalanceFcn function handle, specified as a scalar string or string array. By default, the EngineDataList property is empty, indicating that no additional engine data is required.

If a strategy requires engine data, you can specify EngineDataList as a vector of strings containing the names of the required engineData. The backtestEngine creates an engineData struct with a field corresponding to each element in the EngineDataList. Valid engineData values are:

- Weights — The current portfolio weights before rebalancing,specified as decimal percentages. This is the default first argument to the rebalance function (RebalanceFcn).
- WeightsHistory — A timetable of Weights vectors for each past date in the backtest. This timetable holds the end-of-day asset weights for each asset for each day. Future dates in the timetable contain NaN values.
- PortfolioValue — The current portfolio value at the time the rebalance function (RebalanceFcn) is called, specified as a numeric scalar. This PortfolioValue is potentially different from the "end of day" portfolio value reported by the backtestEngine at the end of the backtest. Transaction costs and other fees are paid *after* the rebalance function is called, so the PortfolioValue passed into the rebalance function by the engineData struct may be different from the final portfolio value reported for that day after the backtest completes.
- PortfolioValueHistory — A timetable of portfolio values for the backtest. The timetable contains two columns. The BeforeExpenses column holds the portfolio value for each day before any expenses (transaction costs or fees). The EndOfDay column holds the portfolio value for each day, inclusive of all transaction costs and other fees. The values in the 2nd column (EndOfDay) match the final end-of-day portfolio values reported at the end of the backtest. Future dates in the timetable contain NaN values.
- FeesHistory — A timetable of fees charged for each day in the backtest. The FeesHistory timetable has two columns: Management and Performance for the management and performance fees, respectively. The management and performance fees are defined using the ManagementFee and PerformanceFee name-value arguments. Future dates in the timetable contain NaN values.

For an example of using EngineDataList, see “Use EngineDataList Property to Enforce Trading Strategy of Whole Shares” on page 15-222.

Data Types: string

Properties

Name — Strategy name

string

Strategy name, specified as a string.

Data Types: string

RebalanceFcn — Rebalance function

function handle

Rebalance function, specified as a function handle.

Data Types: `function_handle`

`RebalanceFrequency` — Rebalance frequency during backtest
`1` (default) | numeric | `duration` object | `calendarDuration` object | vector of datetimes

Rebalance frequency during the backtest, specified as a scalar numeric, `duration` or `calendarDuration` object, or vector of datetimes.

Data Types: `double` | `object` | `datetime`

`TransactionCosts` — Transaction costs
`0` (default) | numeric | vector | function handle

Transaction costs, specified as a scalar numeric, vector, or function handle.

Data Types: `double` | `function_handle`

`LookbackWindow` — Lookback window
`[0 Inf]` (default) | numeric | vector

Lookback window, specified as a scalar numeric or vector.

Data Types: `double`

`InitialWeights` — Initial weights
`[ ]` (default) | vector

Initial weights, specified as a vector.

Data Types: `double`

`ManagementFee` — Management fee charged as an annualized percent of the total portfolio value
`0` (default) | decimal

Management fee charged as an annualized percent of the total portfolio value, specified as a numeric scalar.

Data Types: `double`

`ManagementFeeSchedule` — Management fee schedule
`calyears(1)` (default) | integer | vector of datetimes | `duration` object | `calendarDuration` object

Management fee schedule, specified as scalar numeric, a vector of datetimes, or a `duration` or `calendarDuration` object.

Data Types: `double` | `datetime` | `object`

`PerformanceFee` — Performance fee charged as an absolute percent of the total portfolio growth
`0` (default) | decimal

Performance fee charged as an absolute percent of the total portfolio growth, specified as a numeric scalar.

Data Types: `double`

`PerformanceHurdle` — Annualized hurdle rate or column in assets or signals to act as hurdle asset

`0` (default) | decimal | string

Annualized hurdle rate or column in `assetPrices` or `signalData` to act as hurdle asset, specified as a numeric scalar or a string in either `assetPrices` or `signalData` as a hurdle asset.

Data Types: `double` | `string`

`PerformanceFeeSchedule` — Performance fee schedule

`calyears(1)` (default) | decimal | vector of datetimes | `duration` object | `calendarDuration` object

Performance fee schedule, specified as a scalar numeric, vector of datetimes, or a `duration` or `calendarDuration` object.

Data Types: `datetime` | `object`

`UserData` — Strategy-specific user data for use in `rebalanceFcn`

`[]` (default) | struct

Strategy-specific user data for use in `rebalanceFcn`, specified as a struct.

Data Types: `struct`

`EngineDataList` — Specify set of optional backtest state data a strategy needs in `rebalanceFcn`

`[]` (default) | string

Specify set of optional backtest state data a strategy needs in `rebalanceFcn`, specified as a scalar string or string array.

Data Types: `string`

Examples

Create Backtesting Strategies

Define a backtest strategy by using a `backtestStrategy` object. `backtestStrategy` objects contain properties specific to a trading strategy, such as the rebalance frequency, transaction costs, and a rebalance function. The rebalance function implements the core logic of the strategy and is used by the backtesting engine during the backtest to allow the strategy to change its asset allocation and to make trades. In this example, to illustrate how to create and use backtest strategies in MATLAB®, you prepare two simple strategies for backtesting:

1. An equal weighted strategy
2. A strategy that attempts to "chase returns"

The strategy logic for these two strategies is defined in the rebalance functions on page 15-218.

Set Strategy Properties

A `backtestStrategy` object has several properties that you set using parameters for the `backtestStrategy` function.

Initial Weights

The InitialWeights property contains the asset allocation weights at the start of the backtest. The default value for InitialWeights is empty ([]), which indicates that the strategy begins the backtest uninvested, meaning that 100% of the capital is in cash earning the risk-free rate.

Set the InitialWeights to a specific asset allocation. The size of the initial weights vector must match the number of assets in the backtest.

```
% Initialize the strategies with 30 weights, since the backtest
% data comes from a year of the 30 DJIA stocks.
numAssets = 30;

% Give the initial weights for both strategies equal weighting. Weights
% must sum to 1 to be fully invested.
initialWeights = ones(1,numAssets);
initialWeights = initialWeights / sum(initialWeights);
```

Transaction Costs

The TransactionCosts property allows you to set the fees that the strategy pays for trading assets. Transaction costs are paid as a percentage of the total change in position for each asset. Specify costs in decimal percentages. For example, if TransactionCosts is set to 1% (0.01) and the strategy buys $100 worth of a stock, then the transaction costs incurred are $1.

Transaction costs are set using a 1-by-2 vector that sets separate fee rates for purchases and sales of assets. In this example, both strategies pay the same transaction costs — 25 basis points for asset purchases and 50 basis points for sales.

```
% Define the Transaction costs as [buyCosts sellCost] and specify the costs
% as decimal percentages.
tradingCosts = [0.0025 0.005];
```

You can also set the TransactionCosts property to a function handle if you need to implement arbitrarily complex transaction cost structures. For more information on creating transaction cost functions, see backtestStrategy.

Rebalance Frequency

The RebalanceFrequency property determines how often the backtesting engine rebalances and reallocates the portfolio of a strategy using the rebalance function. Set the RebalanceFrequency in terms of time steps in the backtest. For example, if the backtesting engine is testing a strategy with a set of daily price data, then set the rebalance function in days. Essentially, RebalanceFrequency represents the number of rows of price data to process between each call to the strategy rebalance function.

```
% Both strategies rebalance every 4 weeks (20 days).
rebalFreq = 20;
```

Lookback Window

Each time the backtesting engine calls a strategy rebalance function, a window of asset price data (and possibly signal data) is passed to the rebalance function. The rebalance function can then make trading and allocation decisions based on a rolling window of market data. The LookbackWindow property sets the size of these rolling windows. Set the window in terms of time steps. The window determines the number of rows of data from the asset price timetable that are passed to the rebalance function.

The LookbackWindow property can be set in two ways. For a fixed-sized rolling window of data (for example, "50 days of price history"), the LookbackWindow property is set to a single scalar value (N = 50). The software then calls the rebalance function with a price timetable containing exactly N rows of rolling price data.

Alternatively, you can define the LookbackWindow property by using a 1-by-2 vector [min max] that specifies the minimum and maximum size for an expanding window of data. In this way, you can set flexible window sizes. For example:

- [10 Inf] — At least 10 rows of data
- [0 50] — No more than 50 rows of data
- [0 Inf] — All available data (that is, no minimum, no maximum); this is the default value
- [20 20] — Exactly 20 rows of data; this is equivalent to setting LookbackWindow to the scalar value 20

The software does not call the rebalance function if the data is insufficient to create a valid rolling window, regardless of the value of the RebalanceFrequency property.

If the strategy does not require any price or signal data history, then you can indicate that the rebalance function requires no data by setting the LookbackWindow property to 0.

```
% The equal weight strategy does not require any price history data.
ewLookback = 0;

% The "chase returns" strategy bases its decisions on the trailing
% 10-day asset returns. The lookback window is set to 11 since computing 10 days
% of returns requires the close price from day 0.
chaseLookback = 11;
```

Rebalance Function

The rebalance function (rebalanceFcn) is the user-authored function that contains the logic of the strategy. The backtesting engine calls the strategy rebalance function with a fixed set of parameters and expects it to return a vector of asset weights representing the new, desired portfolio allocation after a rebalance. For more information, see the rebalance functions on page 15-218.

Create Strategies

Using the prepared strategy properties, you can create the two strategy objects.

```
% Create the equal weighted strategy. The rebalance function @equalWeights
% is defined in the Rebalance Functions section at the end of this example.
equalWeightStrategy = backtestStrategy("EqualWeight",@equalWeight, ...
    'RebalanceFrequency',rebalFreq, ...
    'TransactionCosts',tradingCosts, ...
    'LookbackWindow',ewLookback, ...
    'InitialWeights',initialWeights)
```

```
equalWeightStrategy = 
  backtestStrategy with properties:

                  Name: "EqualWeight"
          RebalanceFcn: @equalWeight
    RebalanceFrequency: 20
      TransactionCosts: [0.0025 0.0050]
        LookbackWindow: 0
```

```
            InitialWeights: [0.0333 0.0333 0.0333 0.0333 0.0333 0.0333 0.0333 0.0333 0.0333 0.03
             ManagementFee: 0
     ManagementFeeSchedule: 1y
            PerformanceFee: 0
    PerformanceFeeSchedule: 1y
        PerformanceHurdle: 0
                  UserData: [0x0 struct]
            EngineDataList: [0x0 string]
```

```
% Create the "chase returns" strategy.  The rebalance function
% @chaseReturns is defined in the Rebalance Functions section at the end of this example.
chaseReturnsStrategy = backtestStrategy("ChaseReturns",@chaseReturns, ...
    'RebalanceFrequency',rebalFreq, ...
    'TransactionCosts',tradingCosts, ...
    'LookbackWindow',chaseLookback, ...
    'InitialWeights',initialWeights)
```

```
chaseReturnsStrategy = 
  backtestStrategy with properties:

                      Name: "ChaseReturns"
              RebalanceFcn: @chaseReturns
        RebalanceFrequency: 20
          TransactionCosts: [0.0025 0.0050]
            LookbackWindow: 11
            InitialWeights: [0.0333 0.0333 0.0333 0.0333 0.0333 0.0333 0.0333 0.0333 0.0333 0.03
             ManagementFee: 0
     ManagementFeeSchedule: 1y
            PerformanceFee: 0
    PerformanceFeeSchedule: 1y
        PerformanceHurdle: 0
                  UserData: [0x0 struct]
            EngineDataList: [0x0 string]
```

Set Up Backtesting Engine

To backtest the two strategies, use the `backtestEngine` object. The backtesting engine sets parameters of the backtest that apply to all strategies, such as the risk-free rate and initial portfolio value. For more information, see `backtestEngine`.

```
% Create an array of strategies for the backtestEngine.
strategies = [equalWeightStrategy chaseReturnsStrategy];

% Create backtesting engine to test both strategies.
backtester = backtestEngine(strategies);
```

Rebalance Functions

Strategy rebalance functions defined using the `rebalanceFcn` argument for `backtestStrategy` must adhere to a fixed API that the backtest engine expects when interacting with each strategy. Rebalance functions must implement one of the following two syntaxes:

```
function new_weights = exampleRebalanceFcn(current_weights,assetPriceTimeTable)

function new_weights = exampleRebalanceFcn(current_weights,assetPriceTimeTable,signalDataTimeTab
```

All rebalance functions take as their first input argument the current allocation weights of the portfolio. current_weights represents the asset allocation just before the rebalance occurs. During a rebalance, you can use current_weights in a variety of ways. For example, you can use current_weights to determine how far the portfolio allocation has drifted from the target allocation or to size trades during the rebalance to limit turnover.

The second and third arguments of the rebalance function syntax are the rolling windows of asset prices and optional signal data. The two tables contain the trailing N rows of the asset and signal timetables that are passed to the runBacktest function, where N is set using the LookbackWindow property of each strategy.

If optional signal data is provided to the runBacktest function, then the backtest engine passes the rolling window of signal data to each strategy that supports it.

The equalWeight strategy simply invests equally across all assets.

```
function new_weights = equalWeight(current_weights,assetPrices) %#ok<INUSD>

% Invest equally across all assets.
num_assets = numel(current_weights);
new_weights = ones(1,num_assets) / num_assets;

end
```

The chaseReturns strategy invests only in the top *X* stocks based on their rolling returns in the lookback window. This naive strategy is used simply as an illustrative example.

```
function new_weights = chaseReturns(current_weights,assetPrices)

% Set number of stocks to invest in.
numStocks = 15;

% Compute rolling returns from lookback window.
rollingReturns = assetPrices{end,:} ./ assetPrices{1,:};

% Select the X best performing stocks over the lookback window
[~,idx] = sort(rollingReturns,'descend');
bestStocksIndex = idx(1:numStocks);

% Initialize new weights to all zeros.
new_weights = zeros(size(current_weights));

% Invest equally across the top performing stocks.
new_weights(bestStocksIndex) = 1;
new_weights = new_weights / sum(new_weights);

end
```

Use UserData Property to Create Stop Loss Trading Strategy

This example shows how to use the UserData property of backtestStrategy to create a stop loss trading strategy. A stop loss strategy sets a threshold to define how much money that you are willing to lose before closing a position. For example, if you bought a stock at $100, you could set a stop loss at $95, and if the stock price falls below that point, then you sell your position. You can then set a new price that is your buy-in price to buy back into the stock.

Load Data

```
% Load equity adjusted price data and convert to timetable
T = readtable('dowPortfolio.xlsx');
pricesTT = table2timetable(T(:,[1 3:end]),'RowTimes','Dates')
```

```
pricesTT=251×30 timetable
       Dates         AA       AIG      AXP      BA        C       CAT      DD       DIS      GE
    ___________    _____    _____    _____    _____    _____    _____    _____    _____    _____

    03-Jan-2006    28.72    68.41    51.53    68.63    45.26    55.86    40.68    24.18     33.6
    04-Jan-2006    28.89    68.51    51.03    69.34    44.42    57.29    40.46    23.77    33.56
    05-Jan-2006    29.12     68.6    51.57    68.53    44.65    57.29    40.38    24.19    33.47
    06-Jan-2006    29.02    68.89    51.75    67.57    44.65    58.43    40.55    24.52     33.7
    09-Jan-2006    29.37    68.57    53.04    67.01    44.43    59.49    40.32    24.78    33.61
    10-Jan-2006    28.44    69.18    52.88    67.33    44.57    59.25     40.2    25.09    33.43
    11-Jan-2006    28.05     69.6    52.59     68.3    44.98    59.28    38.87    25.33    33.66
    12-Jan-2006    27.68    69.04     52.6     67.9    45.02    60.13    38.02    25.41    33.25
    13-Jan-2006    27.81    68.84     52.5     67.7    44.92    60.24    37.86    25.47    33.35
    17-Jan-2006    27.97    67.84    52.03    66.93    44.47    60.85    37.75    25.15     33.2
    18-Jan-2006    27.81    67.42    51.84    66.58    44.41    60.04    37.54    24.97    33.08
    19-Jan-2006    28.33    66.92    51.66    66.42    44.02    60.66    37.69       26    32.95
    20-Jan-2006    27.67    65.55    50.49    64.79    41.95    58.98    37.34    25.49     31.7
    23-Jan-2006    28.03    65.46    50.53     65.3    42.24    59.23    37.37    25.29    31.63
    24-Jan-2006    28.16    65.39    51.89    65.92    42.25    59.53    37.09    25.76    31.32
    25-Jan-2006    28.57    64.67    51.97    65.19    42.45    60.23    37.05    25.21    31.13
      ⋮
```

Create Backtest Stop Loss Strategy

The stop loss strategy will either be 100% invested in the stock or 100% in cash. You can use two fields in the UserData struct for backtestStrategy to set the stop loss limits:

- StopLossPercent — How much you are willing to lose before selling, as a decimal percent
- BuyInPercent — How much *further* a stock must fall after you sell it before we buy back in, as a percent

In addition, the UserData struct has the following fields:

- Asset — Ticker symbol AIG for the asset you want to trade
- StopLossPrice — Threshold to sell the asset. If the price falls below this stop loss price, sell the asset and go to cash.
- BuyInPrice — Target price to buy the asset. If the asset is at this price or lower, buy the asset.

```
stopLossStrat = backtestStrategy("StopLoss",@stopLossSingleAsset,RebalanceFrequency=1);
```

```
% Setup the initial UserData
stopLossStrat.UserData = struct(StopLossPercent=0.05,BuyInPercent=0.02,Asset="AIG",StopLossPrice=
```

The backtest strategy starts 100% in cash. Normally the BuyInPrice is set by the strategy after you sell a stock, but in this case, you can initialize the BuyInPrice to be Inf. This triggers a stock buy on the first rebalance (since any price is less than Inf). You don't need to set the StopLossPrice at this point (it is set to nan) because the strategy sets the stop loss price once a stock purchase is made.

The strategy specifies that you are only willing to lose 5% of your capital before the stop loss is triggered and you sell. Then you will buy back into the stock if it falls by an additional 2%. You can set the rebalance frequency to `1` because this type of strategy has to "rebalance" every day in order to check the price to determine if any action is needed.

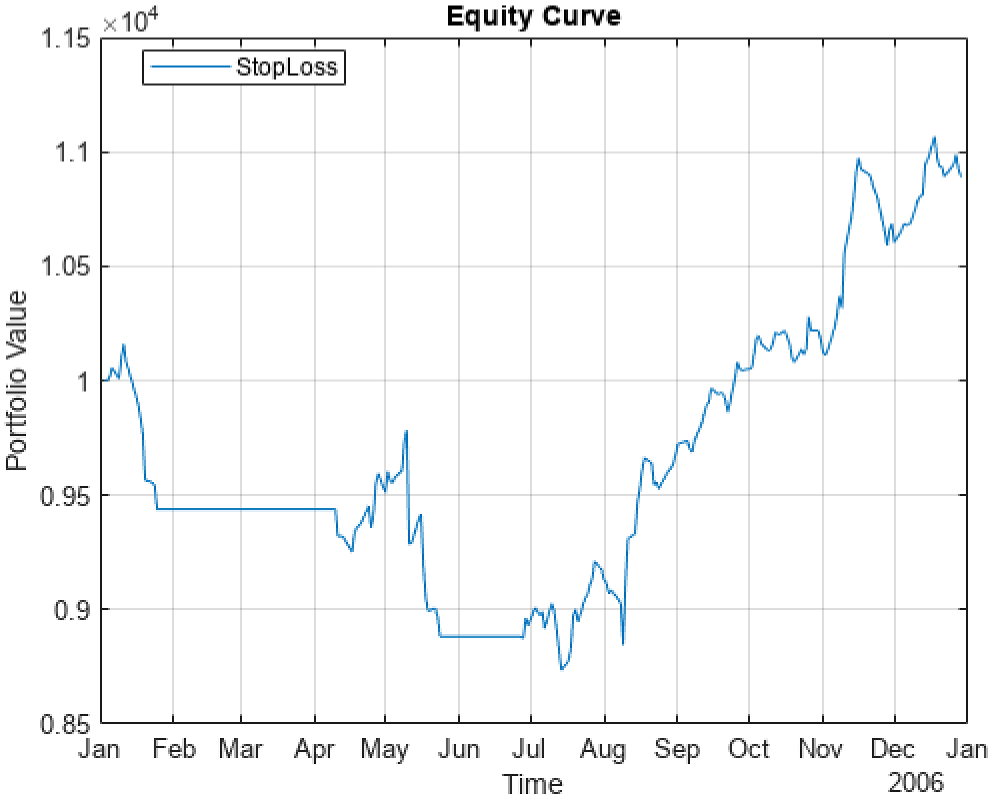
Run Backtest

Create a `backtestEngine` object and run the backtest using `runBacktest`.

```
backtester = backtestEngine(stopLossStrat);
backtester = runBacktest(backtester,pricesTT);
```

Use `equityCurve` to generate a plot the stop loss trading strategy.

```
equityCurve(backtester)
```



The flat parts of the equity curve are where the stop loss strategy has sold AIG and then later buys back into AIG.

Local Functions

```
function [new_weights,user_data] = incrementCounter(current_weights,prices,user_data)

% don't mess with the weights
new_weights = current_weights;

% increment the user data counter
user_data.Counter = user_data.Counter + 1;
```

```
end

function [new_weights,user_data] = stopLossSingleAsset(current_weights,prices,user_data)

% start with the existing weights
new_weights = current_weights;

% find column of the asset
asset = char(user_data.Asset);
assetIdx = find(strncmpi(asset,prices.Properties.VariableNames,numel(asset)));
assetPrice = prices{end,assetIdx};

% determine if we're currently invested or in cash
inCash = sum(current_weights) < 1e-5;

if inCash

    % We are in cash, see if we should buy back in
    if assetPrice <= user_data.BuyInPrice

        % Buy back in
        new_weights(assetIdx) = 1;
        % Set new stop loss price
        user_data.StopLossPrice = assetPrice * (1 - user_data.StopLossPercent);
    end

else

    % We are in the stock, see if we should sell
    if assetPrice <= user_data.StopLossPrice

        % Sell the stock
        new_weights(assetIdx) = 0;
        % Set new buy-in price
        user_data.BuyInPrice = assetPrice * (1 - user_data.BuyInPercent);
    end

end

end
```

Use EngineDataList Property to Enforce Trading Strategy of Whole Shares

This example shows how to use the `EngineDataList` property of `backtestStrategy` to enforce a trading strategy with a whole shares constraint.

Load Data

```
% Load equity adjusted price data and convert the data to timetable
T = readtable('dowPortfolio.xlsx');
pricesTT = table2timetable(T(:,[1 3:end]),'RowTimes','Dates')
```

```
pricesTT=251×30 timetable
        Dates         AA       AIG      AXP      BA        C       CAT      DD       DIS      GE
    ___________     ______   ______   ______   ______   ______   ______   ______   ______   ______
    03-Jan-2006    28.72    68.41    51.53    68.63    45.26    55.86    40.68    24.18     33.6
    04-Jan-2006    28.89    68.51    51.03    69.34    44.42    57.29    40.46    23.77    33.56
    05-Jan-2006    29.12     68.6    51.57    68.53    44.65    57.29    40.38    24.19    33.47
    06-Jan-2006    29.02    68.89    51.75    67.57    44.65    58.43    40.55    24.52     33.7
    09-Jan-2006    29.37    68.57    53.04    67.01    44.43    59.49    40.32    24.78    33.61
    10-Jan-2006    28.44    69.18    52.88    67.33    44.57    59.25     40.2    25.09    33.43
    11-Jan-2006    28.05     69.6    52.59     68.3    44.98    59.28    38.87    25.33    33.66
    12-Jan-2006    27.68    69.04     52.6     67.9    45.02    60.13    38.02    25.41    33.25
    13-Jan-2006    27.81    68.84     52.5     67.7    44.92    60.24    37.86    25.47    33.35
    17-Jan-2006    27.97    67.84    52.03    66.93    44.47    60.85    37.75    25.15     33.2
    18-Jan-2006    27.81    67.42    51.84    66.58    44.41    60.04    37.54    24.97    33.08
    19-Jan-2006    28.33    66.92    51.66    66.42    44.02    60.66    37.69       26    32.95
    20-Jan-2006    27.67    65.55    50.49    64.79    41.95    58.98    37.34    25.49     31.7
    23-Jan-2006    28.03    65.46    50.53     65.3    42.24    59.23    37.37    25.29    31.63
    24-Jan-2006    28.16    65.39    51.89    65.92    42.25    59.53    37.09    25.76    31.32
    25-Jan-2006    28.57    64.67    51.97    65.19    42.45    60.23    37.05    25.21    31.13
         ⋮
```

Create Backtest Strategies

This example runs a backtest on two equal-weighted strategies and then compares the results. The first backtest strategy uses an *exact* equal-weight rebalance function that allows fractional shares. The second backtest strategy uses a `EngineDataList` property for `PortfolioValue` to enforce a *whole shares* constraint.

Use `backtestStrategy` to create the first strategy for partial shares.

```
% Partial shares and equal weight for assets
partialRebal = @(~,prices) ones(1,width(prices)) / width(prices);
ewPartial = backtestStrategy("PartialShares",partialRebal,RebalanceFrequency=1);
```

Use `backtestStrategy` to create the second strategy for whole shares.

```
% Whole shares and equal weight for assets
ewWhole = backtestStrategy("WholeShares",@equalWeightShares,RebalanceFrequency=1,EngineDataList=
```

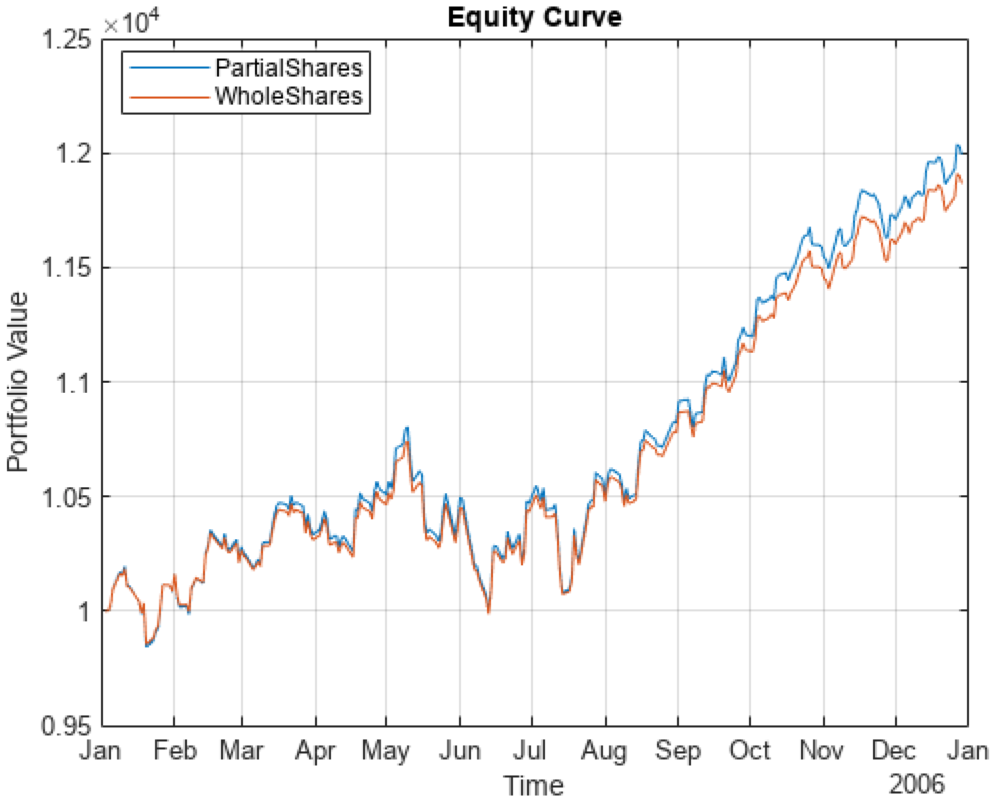
Run Backtest

Aggregate the two strategy objects, create a `backtestEngine` object, and then run the backtest using `runBacktest`.

```
strats = [ewPartial, ewWhole];
backtester = backtestEngine(strats,RiskFreeRate=0.02);
backtester = runBacktest(backtester,pricesTT);
```

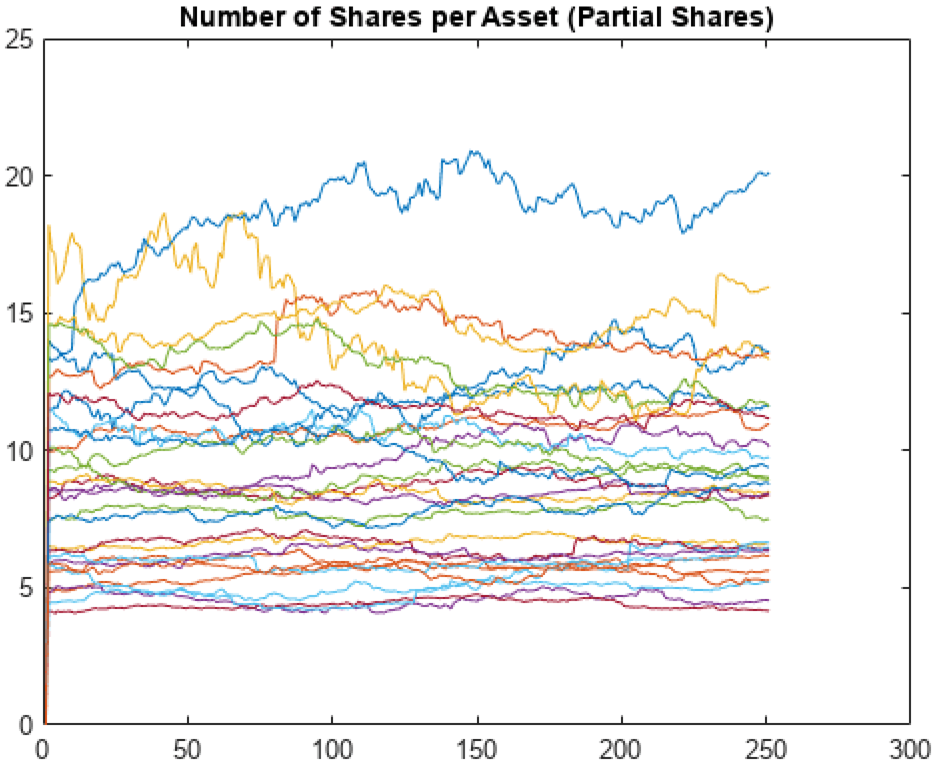
Use `equityCurve` to generate a plot for both of the trading strategies.

```
equityCurve(backtester);
```

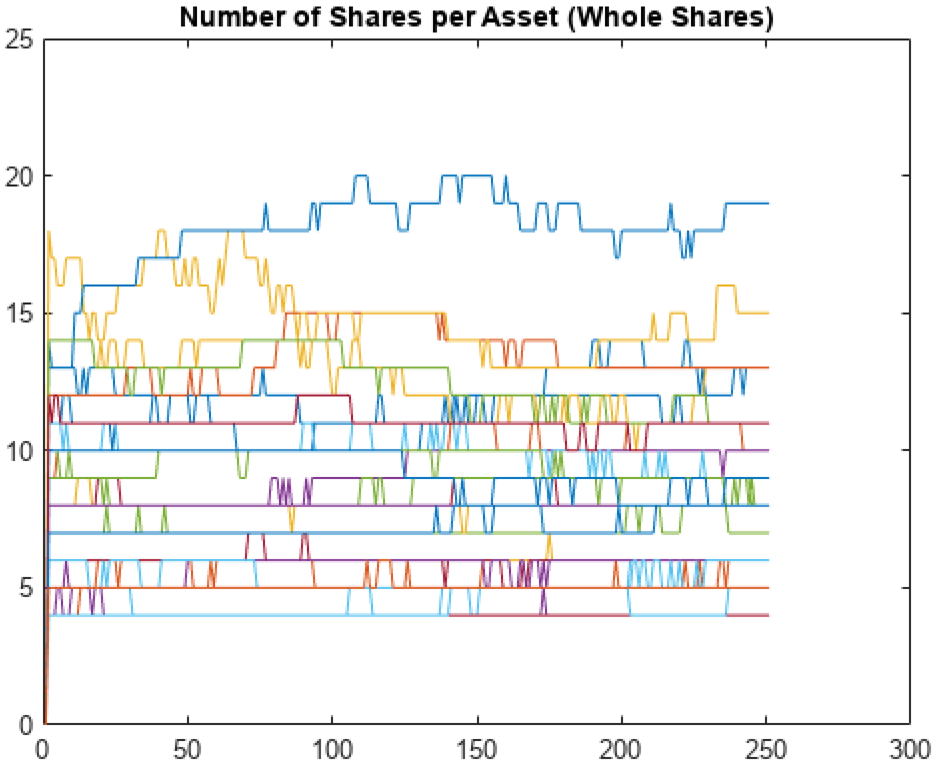


You can verify that the second strategy traded in whole shares.

```
% Plot the number of shares of each asset
plot(backtester.Positions.PartialShares{:,2:end} ./ pricesTT{:,:});
title('Number of Shares per Asset (Partial Shares)')
```



```
plot(backtester.Positions.WholeShares{:,2:end} ./ pricesTT{:,:})
title('Number of Shares per Asset (Whole Shares)')
```



Local Functions

```
function new_weights = equalWeightShares(engine_data,prices)

numAssets = width(prices);

% Dollars per asset for equal weight with partial shares
dollars_per_asset = engine_data.PortfolioValue / numAssets;
% Compute shares (partial shares first, then round down)
asset_shares = floor(dollars_per_asset ./ prices{end,:});
% Convert number of shares into dollars
asset_dollars = asset_shares .* prices{end,:};
% Convert dollars into weights
new_weights = asset_dollars / engine_data.PortfolioValue;

end
```

More About

Defining Schedules for Backtest Strategies

A backtesting schedule is a set of dates defining when specific events will occur.

The `backtestStrategy` object uses schedules in several of ways, including setting the rebalance frequency (`RebalanceFrequency`), the performance fee (`PerformanceFeeSchedule`), and the management fee (`ManagementFeeSchedule`) schedules. Schedules are specified with a syntax that supports a variety of data types, which include numeric scalars, a vector of datetimes, and duration or calendarDuration objects. The resulting schedules for the different data types are:

- Numeric — If a schedule is defined using a numeric scalar, it refers to the number of rows (of the assetPrices timetable) between each event in the schedule, starting from the backtest start date. For example, if a schedule is set to 10, then the schedule event (a rebalance date or fee payment date) occurs on every 10th row of the prices timetable while the backtest runs.
- duration or calendarDuration — If specified as a duration or calendarDuration object, the schedule specifies a duration of time between each event (rebalance date or fee payment date). The schedule is calculated starting at the backtest start date and then schedule dates are added after each step of the specified duration.
- Vector of datetimes — If the numeric or duration syntaxes are not appropriate, then you can explicitly specify schedules using a vector of datetimes.

Note For both the duration and datetime syntaxes, if a resulting schedule date is not found in the backtest data set (the assetPrices timetable), the backtestEngine either adjusts the date to the nearest valid date or else issues an error. This behavior is controlled by the DateAdjustment property of the backtestEngine object. By default, dates are adjusted to the nearest previous date. For example, if a schedule date falls on a Saturday, and is not found in the backtest data set, then the date is adjusted to the previous Friday. Alternatively, the backtestEngine DateAdjustment property allows you to adjust dates to the next valid date, or to issue an error if a date is not found by requiring exact dates.

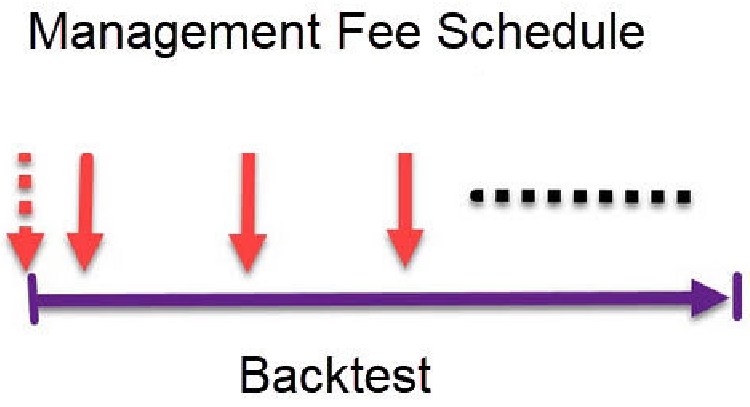
Management Fees

The management fee is the annualized rate charged on the assets under management of the strategy to pay for the fund's management costs.

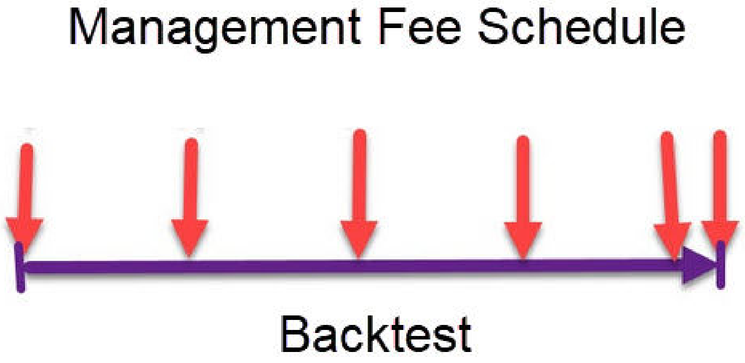
The fee is charged on each date specified by the ManagementFeeSchedule name-value argument. For more information, see “Defining Schedules for Backtest Strategies” on page 15-226.

The management fee is forward looking, meaning that a fee paid on a given date covers the management of the strategy from that date to the next management fee date. For some syntaxes of theManagementFeeSchedule, the resulting schedule does not begin and end precisely on the backtest start and end dates. In these cases, the backtest engine adds default dates to the start and end of the schedule to ensure that the entire backtest is covered by management fees. For example:

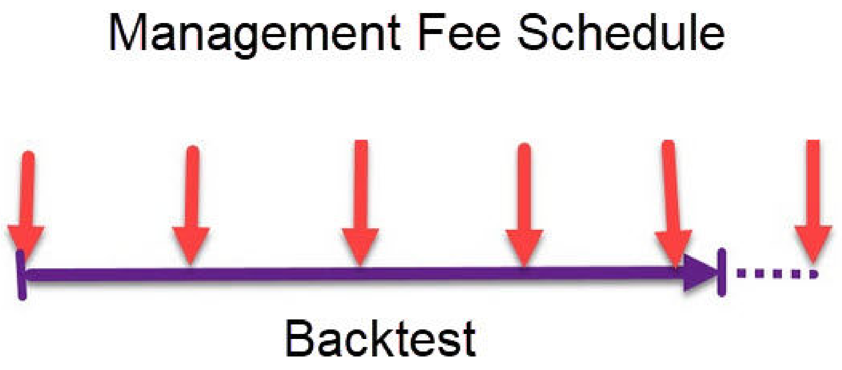
- If ManagementFeeSchedule is specified as a vector of datetimes, and the backtest start date is not included in the vector, then the start date is added to the fee schedule.



- If `ManagementFeeSchedule` is specified as a numeric scalar or a vector of datetimes, and the final specified fee date occurs before the backtest end date, then the backtest end date is added to the fee schedule.



- If `ManagementFeeSchedule` is specified as a `duration` or `calendarDuration` object, and the backtest end date is not included in the generated schedule, then a final fee date is added to the end by adding the duration object to the (previous) final fee date. This adjustment can produce a final fee date that is beyond the end of the backtest, but it results in more realistic fees paid at the final payment date.



For more information on where the resulting management fees are reported, see the `backtestEngine` object read-only property "Fees" on page 15-0 .

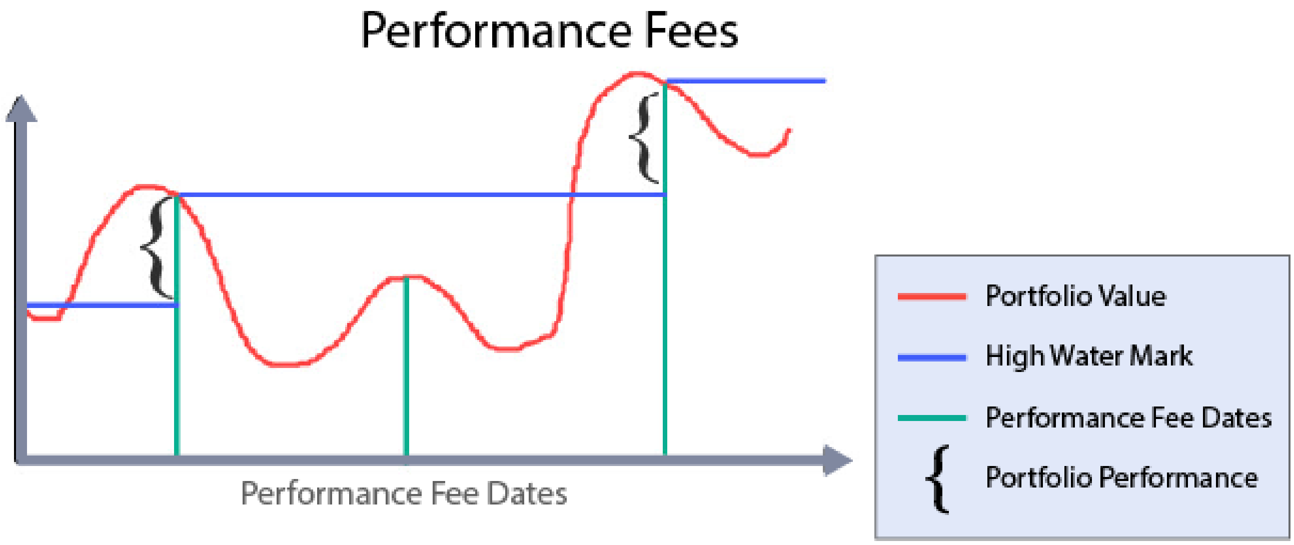
Performance Fees

A performance fee, or incentive fee, is a periodic payment that is paid by fund investors based on the fund's performance.

The performance fee can have a variety of structures, but it often is calculated as a percentage of a fund's *increase* in NAV over some tracked high-water mark. This calculation ensures the fee is paid only when the investment manager produces positive results. Performance fees in hedge funds are sometimes set at 20%, meaning if the fund NAV goes from $1M to $1.2M, a fee of $40k would be charged (20% of the $200k growth).

During the backtest, the backtest engine tracks a high-water mark. The high-water mark starts at the portfolio initial value. At each performance fee date, if the portfolio value is greater than the previous high-water mark, then the high-water mark is updated to the new portfolio value and the performance fee is paid on the growth of the portfolio. The portfolio growth is the difference between the new high-water mark and the previous one. If, on a performance fee date, the portfolio value is less than the previous high-water mark, then no fee is paid and the high-water mark is not updated.

The high-water mark is typically updated at each performance fee date and does not track maximum portfolio values between dates, as illustrated:



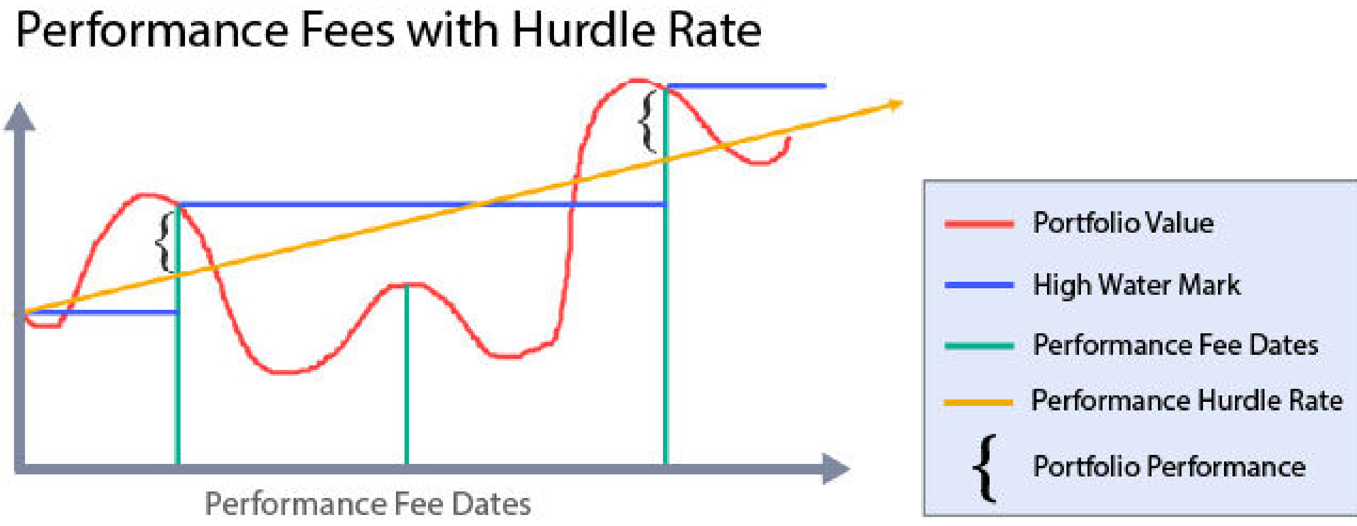
For more information on where the resulting performance fees are reported, see the `backtestEngine` object read-only property “Fees” on page 15-0 .

Performance Hurdle

A hurdle is an optional feature of a performance fee.

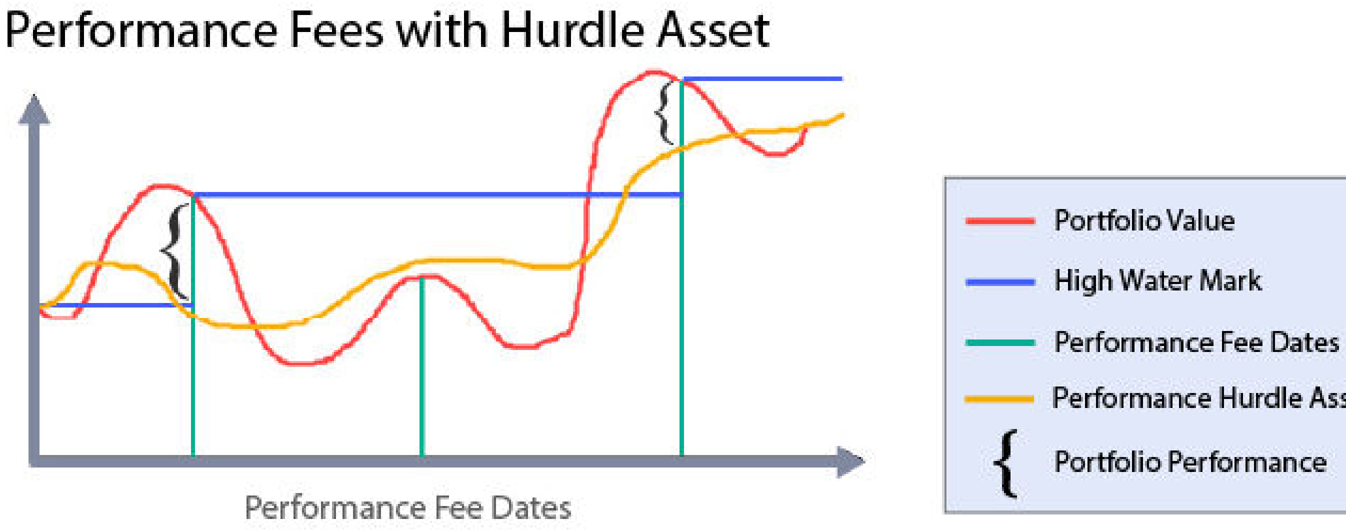
A hurdle sets a minimum performance threshold that a fund must beat before the performance fee is charged. Hurdles can be fixed rates of return, a *hurdle rate*, such as 5%. For example, if the fund NAV goes from $1M to $1.2M, the performance fee is charged only on the performance beyond 5%. A 5% hurdle rate on $1M sets the target hurdle value at $1.05M. The $1.2M portfolio is $150k over the hurdle value. The performance fee is charged only on that $150k, so 20% of $150k is a $30k performance fee.

The following figure illustrates a 5% hurdle rate.



You can also specify a performance hurdle as a benchmark asset or index. In this case, the performance fee is paid only on the fund performance in excess of the hurdle instrument. For example, assume that the fund NAV goes from $1M to $1.2M and you have specified the S&P500 as your hurdle. If the S&P500 index returns 25% over the same period, then no performance fee is paid because the fund did not produce returns in excess of the hurdle instrument. If the fund value is greater than the hurdle, but still less than the previous high-water mark, again, no performance fee is paid.

At each performance fee date, the fund charges the performance fee on the difference between the current fund value and the maximum of the high-water mark and the hurdle value. The high-water mark is updated only if a performance fee is paid.



For more information on where the resulting performance fees are reported, see the `backtestEngine` object read-only property “Fees” on page 15-0 .

Version History

Introduced in R2020b

R2023a: Support for data sharing in user-defined function handles

The `backtestStrategy` name-value argument for `UserData` supports strategy-specific user data for use in the user-defined function handles for the rebalance function. Also the `backtestStrategy` name-value argument for `EngineDataList` enables you to specify the set of optional backtest state data that a strategy needs in the user-defined function handles.

R2022b: Management and performance fees

The `backtestStrategy` object supports name-value arguments for `ManagementFee`, `ManagementFeeSchedule`, `PerformanceFee`, `PerformanceHurdle`, and `PerformanceFeeSchedule`.

R2022a: Include NaN values in asset price data

Behavior changed in R2022a

The `backtestStrategy` object supports `NaN`s in the `assetPrices` timetable and `NaN`s and `<missing>` in the `signalData` timetable.

See Also

`runBacktest` | `summary` | `backtestEngine` | `equityCurve` | `timetable` | `duration` | `calendarDuration`

Topics

"Backtest Investment Strategies Using Financial Toolbox™" on page 4-231
"Backtest Investment Strategies with Trading Signals" on page 4-244
"Backtest Using Risk-Based Equity Indexation" on page 4-285
"Backtest with Brinson Attribution to Evaluate Portfolio Performance" on page 4-311

External Websites

Backtesting Strategy Framework in Financial Toolbox (2 min 17 sec)

backtestEngine

Create backtestEngine object to backtest strategies and analyze results

Description

Create a backtestEngine to run a backtest of portfolio investment strategies on historical data.

Use this workflow to develop and run a backtest:

1. Define the strategy logic using a backtestStrategy object to specify how a strategy rebalances a portfolio of assets.
2. Use backtestEngine to create a backtestEngine object that specifies parameters of the backtest.
3. Use runBacktest to run the backtest against historical asset price data and, optionally, trading signal data.
4. Use equityCurve to plot the equity curves of each strategy.
5. Use summary to summarize the backtest results in a table format.

For more detailed information on this workflow, see “Backtest Investment Strategies Using Financial Toolbox™” on page 4-231.

Creation

Syntax

```
backtester = backtestEngine(strategies)
backtester = backtestEngine( ___ ,Name,Value)
```

Description

backtester = backtestEngine(strategies) creates a backtestEngine object. Use the backtestEngine object to backtest the portfolio trading strategies defined in the backtestStrategy objects.

backtester = backtestEngine(___ ,Name,Value) sets properties on page 15-236 using name-value pair arguments and any of the arguments in the previous syntax. You can specify multiple name-value pair arguments. For example, backtester = backtestEngine(strategies,'RiskFreeRate',0.02,'InitialPortfolioValue',1000,'RatesConvention',"Annualized",'Basis',2).

Input Arguments

strategies — Backtest strategies
vector of backtestStrategy objects

Backtest strategies, specified as a vector of backtestStrategy objects. Each backtestStrategy object defines a portfolio trading strategy.

Data Types: `object`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `backtester = backtestEngine(strategies,'RiskFreeRate',0.02,'InitialPortfolioValue',1000,'RatesConvention',"Annualized",'Basis',2)`

`RiskFreeRate` — Risk free rate

`0` (default) | numeric | timetable

Risk free rate, specified as the comma-separated pair consisting of `'RiskFreeRate'` and a scalar numeric or a one-column timetable.

Note If you specify a timetable:

- The dates in the specified `timetable` must include the start and end dates of the backtest.
- The series of dates in the specified timetable between the start and end dates (inclusive) must correspond exactly to the corresponding series of dates in the `assetPrices` timetable.

If `RatesConvention` is `"Annualized"`, then `RiskFreeRate` specifies an annualized rate.

If `RatesConvention` is `"PerStep"`, then the `RiskFreeRate` is a decimal percentage and represents the risk free rate for one time step in the backtest. For example, if the backtest uses daily asset price data, then the `RiskFreeRate` value must be the daily rate of return for cash.

Data Types: `double` | `timetable`

`CashBorrowRate` — Cash borrowing rate

`0` (default) | numeric | timetable

Cash borrowing rate, specified as the comma-separated pair consisting of `'CashBorrowRate'` and a scalar numeric or a one-column timetable.

Note If you specify a timetable:

- The dates in the specified `timetable` must include the start and end dates of the backtest.
- The series of dates in the specified timetable between the start and end dates (inclusive) must correspond exactly to the corresponding series of dates in the `assetPrices` timetable.

The `CashBorrowRate` specifies the rate of interest accrual on negative cash balances (margin) during the backtest.

If `RatesConvention` is `"Annualized"`, then `CashBorrowRate` specifies an annualized rate.

If RatesConvention is "PerStep", then the CashBorrowRate value is a decimal percentage and represents the interest accrual rate for one time step in the backtest. For example, if the backtest is using daily asset price data, then the CashBorrowRate value must be the daily interest rate for negative cash balances.

Data Types: double | timetable

InitialPortfolioValue — Initial portfolio value

10000 (default) | numeric

Initial portfolio value, specified as the comma-separated pair consisting of 'InitialPortfolioValue' and a scalar numeric.

Data Types: double

RatesConvention — Defines how backtest engine uses RiskFreeRate and CashBorrowRate to compute interest

"Annualized" (default) | character vector with value 'Annualized' or 'PerStep' | string with value "Annualized" or "PerStep"

Defines how backtest engine uses RiskFreeRate and CashBorrowRate to compute interest, specified as the comma-separated pair consisting of 'RatesConvention' and a character vector or string.

- 'Annualized' — The rates are treated as annualized rates and the backtest engine computes incremental interest based on the day count convention specified in the Basis property. This is the default.
- 'PerStep' — The rates are treated as per-step rates and the backtest engine computes interest at the provided rates at each step of the backtest.

Data Types: char | string

DateAdjustment — Date handling behavior for rebalance dates that are missing from asset prices timetable

"Previous" (default) | character vector with value 'Previous', 'Next', or 'None' | string with value "Previous", "Next", or "None"

Date handling behavior for rebalance dates that are missing from asset prices timetable, specified as the comma-separated pair consisting of 'DateAdjustment' and a character vector or string.

- 'Previous' — For each rebalance date in the rebalance schedule, the rebalance occurs on the nearest date in the asset timetable that occurs on or before the requested rebalance date. This is the default.
- 'Next' — Move to the next date.
- 'None' — Dates are not adjusted and the backtest engine errors when encountering a rebalance date that does not appear in the asset prices timetable.

Data Types: char | string

Basis — Defines day-count convention when computing interest at RiskFreeRate or CashBorrowRate

0 (default) | numeric values: 0,1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13

Defines the day-count convention when computing interest at the `RiskFreeRate` or `CashBorrowRate`, specified as the comma-separated pair consisting of `'Basis'` and a scalar integer 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 2-16.

Note `Basis` is only used when the `RatesConvention` property is set to `"Annualized"`. If the `RatesConvention` is `"PerStep"`, and `Basis` is set, `backtestEngine` ignores the `Basis` value.

Data Types: `double`

Properties

`Strategies` — Backtest strategies
vector of `backtestStrategy` objects

Backtest strategies, specified as a vector of `backtestStrategy` objects.

Data Types: `object`

`RiskFreeRate` — Risk free rate
`0` (default) | numeric | timetable

Risk free rate, specified as a scalar numeric or timetable.

Data Types: `double`

`CashBorrowRate` — Cash borrowing rate
`0` (default) | numeric | timetable

Cash borrowing rate, specified as a scalar numeric or timetable.

Data Types: `double`

InitialPortfolioValue — Initial portfolio value

10000 (default) | numeric

Initial portfolio value, specified as a scalar numeric.

Data Types: double

AnnualizedRates — Use annualized rates for RiskFreeRate and CashBorrowRate

true (default) | logical with value true or false

Use annualized rates for RiskFreeRate and CashBorrowRate, specified as a scalar logical.

Data Types: logical

DateAdjustment — Date handling behavior for rebalance dates that are missing from asset prices timetable

"Previous" (default) | string with value "Previous", "Next", or "None"

Date handling behavior for rebalance dates that are missing from asset prices timetable, specified as a string.

Data Types: char | string

Basis — Day-count basis of annualized rates for RiskFreeRate and CashBorrowRate

0 (default) | numeric values: 0,1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13

Day-count of annualized rates for RiskFreeRate and CashBorrowRate, specified a scalar integer.

Data Types: double

NumAssets — Number of assets in portfolio universe

[] (default) | numeric

This property is read-only.

Number of assets in the portfolio universe, a numeric. NumAssets is derived from the timetable of adjusted prices passed to runBacktest. NumAssets is empty until you run the backtest using the runBacktest function.

Data Types: double

Returns — Strategy returns

[] (default) | timetable

This property is read-only.

Strategy returns, a NumTimeSteps-by-NumStrategies timetable of strategy returns. Returns are per time step. For example, if you use daily prices with runBacktest, then Returns is the daily strategy returns. Returns is empty until you run the backtest using the runBacktest function.

Data Types: timetable

Positions — Asset positions for each strategy

[] (default) | structure

This property is read-only.

Asset positions for each strategy, a structure containing a NumTimeSteps-by-NumAssets timetable of asset positions for each strategy. For example, if you use daily prices in the runBacktest, then the

`Positions` structure holds timetables containing the daily asset positions. `Positions` is empty until you run the backtest using the `runBacktest` function.

Data Types: `struct`

`Turnover` — Strategy turnover
`[]` (default) | timetable

This property is read-only.

Strategy turnover, a `NumTimeSteps`-by-`NumStrategies` timetable. `Turnover` is empty until you run the backtest using the `runBacktest` function.

Data Types: `timetable`

`BuyCost` — Transaction costs for asset purchases of each strategy
`[]` (default) | timetable

This property is read-only.

Transaction costs for the asset purchases of each strategy, a `NumTimeSteps`-by-`NumStrategies` timetable. `BuyCost` is empty until you run the backtest using the `runBacktest` function.

Data Types: `timetable`

`SellCost` — Transaction costs for asset sales of each strategy
`[]` (default) | timetable

This property is read-only.

Transaction costs for the asset sales of each strategy, a `NumTimeSteps`-by-`NumStrategies` timetable. `SellCost` is empty until you run the backtest using the `runBacktest` function.

Data Types: `timetable`

`Fees` — Paid fees for management and performance fees
`[]` (default) | struct

This property is read-only.

Paid fees for management and performance fees, a struct containing a timetable for each strategy which holds all the fees paid by the strategy. The `Fees` timetable contains an entry for each date where at least one fee was paid. Each column holds the amount paid for a particular type of fee. If no fees are paid, then the `Fees` timetable is empty.

For more information on management and performance fees defined using a `backtestStrategy` object, see "Management Fees" on page 15-227, "Performance Fees" on page 15-229, and "Performance Hurdle" on page 15-230.

Data Types: `timetable`

Object Functions

runBacktest	Run backtest on one or more strategies
summary	Generate summary table of backtest results
equityCurve	Plot equity curves of strategies

Examples

Backtest Strategy Using backtestEngine

Use a backtesting engine in MATLAB® to run a backtest on an investment strategy over a time series of market data. You can define a backtesting engine by using `backtestEngine` object. A `backtestEngine` object sets properties of the backtesting environment, such as the risk-free rate, and holds the results of the backtest. In this example, you can create a backtesting engine to run a simple backtest and examine the results.

Create Strategy

Define an investment strategy by using the `backtestStrategy` function. This example builds a simple equal-weighted investment strategy that invests equally across all assets. For more information on creating backtest strategies, see `backtestStrategy`.

```
% The rebalance function is simple enough that you can use an anonymous function
equalWeightRebalanceFcn = @(current_weights,~) ones(size(current_weights)) / numel(current_weigh

% Create the strategy
strategy = backtestStrategy("EqualWeighted",equalWeightRebalanceFcn,...
    'RebalanceFrequency',20,...
    'TransactionCosts',[0.0025 0.005],...
    'LookbackWindow',0)
```

```
strategy = 
  backtestStrategy with properties:

                    Name: "EqualWeighted"
            RebalanceFcn: @(current_weights,~)ones(size(current_weights))/numel(current_weights
      RebalanceFrequency: 20
        TransactionCosts: [0.0025 0.0050]
          LookbackWindow: 0
          InitialWeights: [1x0 double]
           ManagementFee: 0
   ManagementFeeSchedule: 1y
          PerformanceFee: 0
  PerformanceFeeSchedule: 1y
       PerformanceHurdle: 0
                UserData: [0x0 struct]
          EngineDataList: [0x0 string]
```

Set Backtesting Engine Properties

The backtesting engine has several properties that you set by using parameters to the `backtestEngine` function.

Risk-Free Rate

The `RiskFreeRate` property holds the interest rate earned for uninvested capital (that is, cash). When the sum of portfolio weights is below 1, the remaining capital is invested in cash and earns the risk-free rate. The risk-free rate and the cash-borrow rate can be defined in annualized terms or as explicit "per-time-step" interest rates. The `RatesConvention` property is used to specify how the `backtestEngine` interprets the two rates (the default interpretation is "Annualized"). For this example, set the risk-free rate to 2% annualized.

```
% 2% annualized risk-free rate
riskFreeRate = 0.02;
```

Cash Borrow Rate

The CashBorrowRate property sets the interest accrual rate applied to negative cash balances. If at any time the portfolio weights sum to a value greater than 1, then the cash position is negative by the amount in excess of 1. This behavior of portfolio weights is analogous to borrowing capital on margin to invest with leverage. Like the RiskFreeRate property, the CashBorrowRate property can either be annualized or per-time-step depending on the value of the RatesConvention property.

```
% 6% annualized margin interest rate
cashBorrowRate = 0.06;
```

Initial Portfolio Value

The InitialPortfolioValue property sets the value of the portfolio at the start of the backtest for all strategies. The default is $10,000.

```
% Start backtest with $1M
initPortfolioValue = 1000000;
```

Create Backtest Engine

Using the prepared properties, create the backtesting engine using the backtestEngine function.

```
% The backtesting engine takes an array of backtestStrategy objects as the first argument
backtester = backtestEngine(strategy,...
    'RiskFreeRate',riskFreeRate,...
    'CashBorrowRate',cashBorrowRate,...
    'InitialPortfolioValue',initPortfolioValue)
```

```
backtester = 
  backtestEngine with properties:

               Strategies: [1x1 backtestStrategy]
             RiskFreeRate: 0.0200
           CashBorrowRate: 0.0600
          RatesConvention: "Annualized"
                    Basis: 0
    InitialPortfolioValue: 1000000
           DateAdjustment: "Previous"
                NumAssets: []
                  Returns: []
                Positions: []
                 Turnover: []
                  BuyCost: []
                 SellCost: []
                     Fees: []
```

Several additional properties of the backtesting engine are initialized to empty. The backtesting engine populates these properties, which contain the results of the backtest, upon completion of the backtest.

Load Data and Run Backtest

Run the backtest over daily price data from the 30 component stocks of the DJIA.

```
% Read table of daily adjusted close prices for 2006 DJIA stocks
T = readtable('dowPortfolio.xlsx');

% Remove the DJI index column and convert to timetable
pricesTT = table2timetable(T(:,[1 3:end]),'RowTimes','Dates');
```

Run the backtest using the `runBacktest` function.

```
backtester = runBacktest(backtester,pricesTT)
```

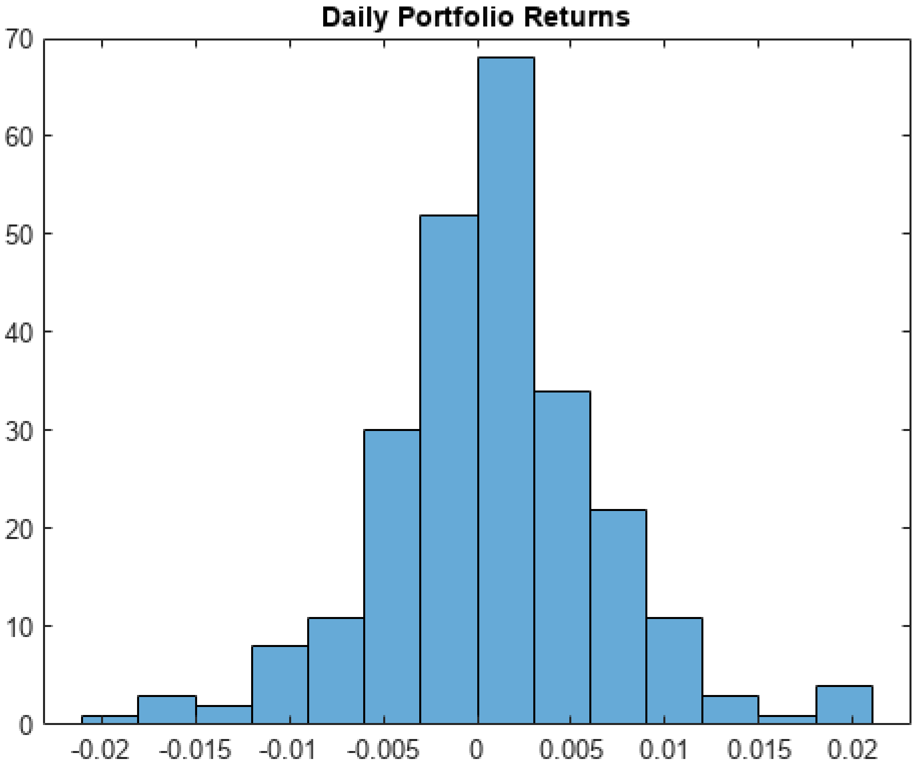
```
backtester = 
  backtestEngine with properties:

               Strategies: [1x1 backtestStrategy]
             RiskFreeRate: 0.0200
           CashBorrowRate: 0.0600
          RatesConvention: "Annualized"
                    Basis: 0
    InitialPortfolioValue: 1000000
           DateAdjustment: "Previous"
                NumAssets: 30
                  Returns: [250x1 timetable]
                Positions: [1x1 struct]
                 Turnover: [250x1 timetable]
                  BuyCost: [250x1 timetable]
                 SellCost: [250x1 timetable]
                     Fees: [1x1 struct]
```

Examine Results

The backtesting engine populates the read-only properties of the `backtestEngine` object with the backtest results. Daily values for portfolio returns, asset positions, turnover, transaction costs, and fees are available to examine.

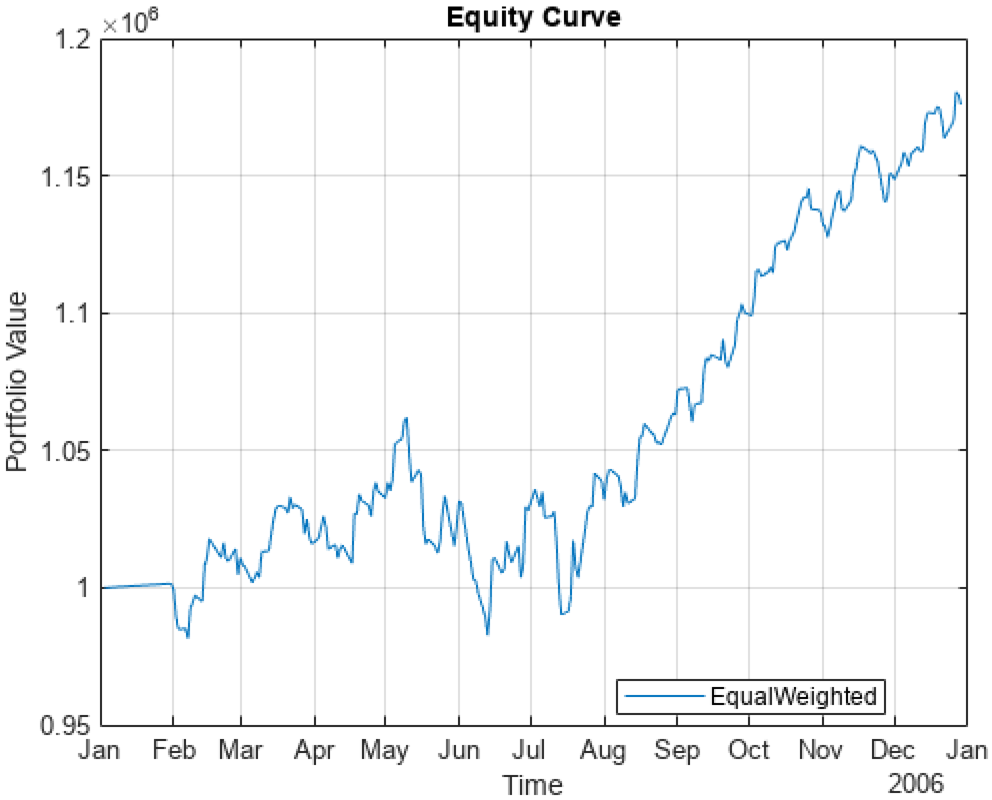
Examine the daily returns.

```
% Generate a histogram of daily portfolio returns
histogram(backtester.Returns{:,1})
title('Daily Portfolio Returns')
```



Use `equityCurve` to plot the equity curve for the simple equal-weighted investment strategy.

```
equityCurve(backtester)
```



Version History

Introduced in R2020b

R2022b: Management and performance fees

Behavior changed in R2022b

The `backtestEngine` object supports a read-only property for `Fees`that reports the management and performance fees paid during a backtest. The `Fees` property is a struct containing a timetable for each strategy and the timetable holds all the fees paid by the strategy.

R2022a: Specify time varying cash rates of return

The `backtestEngine` name-value arguments for `RiskFreeRate` and `CashBorrowRate` support a `timetable` data type.

R2022a: Control how the backtesting framework handles missing rebalance dates

Behavior changed in R2022a

The name-value argument for `DateAdjustment` enables you to control the date handling behavior for rebalance dates that are missing from the `assetPrices` timetable. If a rebalance date falls on a holiday, you can specify the `"Next"` or `"None"` option for `DateAdjustment`.

See Also

`backtestStrategy` | `runBacktest` | `summary` | `equityCurve` | `timetable`

Topics

"Backtest Investment Strategies Using Financial Toolbox™" on page 4-231
"Backtest Investment Strategies with Trading Signals" on page 4-244
"Backtest Using Risk-Based Equity Indexation" on page 4-285
"Backtest with Brinson Attribution to Evaluate Portfolio Performance" on page 4-311

External Websites

Backtesting Strategy Framework in Financial Toolbox (2 min 17 sec)

runBacktest

Run backtest on one or more strategies

Syntax

```
backtester = runBacktest(backtester,pricesTT)
backtester = runBacktest(backtester,pricesTT,signalTT)
backtester = runBacktest( ___ ,Name,Value)
```

Description

`backtester = runBacktest(backtester,pricesTT)` runs the backtest over the timetable of adjusted asset price data.

`runBacktest` initializes each strategy previously defined using `backtestStrategy` to the `InitialPortfolioValue` and then begins processing the timetable of price data (`pricesTT`) as follows:

1. At each time step, the `runBacktest` function applies the asset returns to the strategy portfolio positions.
2. The `runBacktest` function determines which strategies to rebalance based on the `RebalanceFrequency` property of the `backtestStrategy` objects.
3. For strategies that need rebalancing, the `runBacktest` function calls their rebalance functions with a rolling window of asset price data based on the `LookbackWindow` property of each `backtestStrategy`.
4. Transaction costs are calculated and charged based on the changes in asset positions and the `TransactionCosts` property of each `backtestStrategy` object.
5. After the backtest is complete, the results are stored in several properties of the `backtestEngine` object.

`backtester = runBacktest(backtester,pricesTT,signalTT)` run the backtest using the adjusted asset price data and signal data. When you specify the signal data timetable (`signalTT`), then the `runBacktest` function runs the backtest and additionally passes a rolling window of signal data to the rebalance function of each strategy during the rebalance step.

`backtester = runBacktest( ___ ,Name,Value)` specifies options using one or more optional name-value pair arguments in addition to the input arguments in the previous syntax. For example, `backtester = runBacktest(backtester,assetPrices,'Start',50,'End',100)`.

Examples

Run Backtests

The MATLAB® backtesting engine runs backtests of portfolio investment strategies over timeseries of asset price data. After creating a set of backtest strategies using `backtestStrategy` and the backtest engine using `backtestEngine`, the `runBacktest` function executes the backtest. This example illustrates how to use the `runBacktest` function to test investment strategies.

Load Data

Load one year of stock price data. For readability, this example only uses a subset of the DJIA stocks.

```
% Read table of daily adjusted close prices for 2006 DJIA stocks
T = readtable('dowPortfolio.xlsx');

% Prune the table on only hold the dates and selected stocks
timeColumn = "Dates";
assetSymbols = ["BA", "CAT", "DIS", "GE", "IBM", "MCD", "MSFT"];
T = T(:,[timeColumn assetSymbols]);

% Convert to timetable
pricesTT = table2timetable(T,'RowTimes','Dates');

% View the final asset price timetable
head(pricesTT)
```

```
       Dates         BA       CAT      DIS      GE       IBM      MCD      MSFT 
    ___________    _____    _____    _____    _____    _____    _____    _____

    03-Jan-2006    68.63    55.86    24.18     33.6    80.13    32.72    26.19
    04-Jan-2006    69.34    57.29    23.77    33.56    80.03    33.01    26.32
    05-Jan-2006    68.53    57.29    24.19    33.47    80.56    33.05    26.34
    06-Jan-2006    67.57    58.43    24.52     33.7    82.96    33.25    26.26
    09-Jan-2006    67.01    59.49    24.78    33.61    81.76    33.88    26.21
    10-Jan-2006    67.33    59.25    25.09    33.43     82.1    33.91    26.35
    11-Jan-2006     68.3    59.28    25.33    33.66    82.19     34.5    26.63
    12-Jan-2006     67.9    60.13    25.41    33.25    81.61    33.96    26.48
```

Create Strategy

In this introductory example, test an equal weighted investment strategy. This strategy invests an equal portion of the available capital into each asset. This example does describe the details about how create backtest strategies. For more information on creating backtest strategies, see `backtestStrategy`.

Set the `RebalanceFrequency` to rebalance the portfolio every 60 days. This example does not use a lookback window to rebalance.

```
% Create the strategy
numAssets = size(pricesTT,2);
equalWeightsVector = ones(1,numAssets) / numAssets;
equalWeightsRebalanceFcn = @(~,~) equalWeightsVector;

ewStrategy = backtestStrategy("EqualWeighted",equalWeightsRebalanceFcn, ...
    'RebalanceFrequency',60, ...
    'LookbackWindow',0, ...
    'TransactionCosts',0.005, ...
    'InitialWeights',equalWeightsVector)
```

```
ewStrategy = 
  backtestStrategy with properties:

                      Name: "EqualWeighted"
              RebalanceFcn: @(~,~)equalWeightsVector
        RebalanceFrequency: 60
          TransactionCosts: 0.0050
```

```
            LookbackWindow: 0
            InitialWeights: [0.1429 0.1429 0.1429 0.1429 0.1429 0.1429 0.1429]
             ManagementFee: 0
     ManagementFeeSchedule: 1y
            PerformanceFee: 0
    PerformanceFeeSchedule: 1y
         PerformanceHurdle: 0
                  UserData: [0x0 struct]
            EngineDataList: [0x0 string]
```

Run Backtest

Create a backtesting engine and run a backtest over a year of stock data. For more information on creating backtest engines, see `backtestEngine`.

```
% Create the backtest engine. The backtest engine properties that hold the
% results are initialized to empty.
backtester = backtestEngine(ewStrategy)
```

```
backtester = 
  backtestEngine with properties:

               Strategies: [1x1 backtestStrategy]
             RiskFreeRate: 0
           CashBorrowRate: 0
          RatesConvention: "Annualized"
                    Basis: 0
    InitialPortfolioValue: 10000
           DateAdjustment: "Previous"
                NumAssets: []
                  Returns: []
                Positions: []
                 Turnover: []
                  BuyCost: []
                 SellCost: []
                     Fees: []
```

```
% Run the backtest. The empty properties are now populated with
% timetables of detailed backtest results.
backtester = runBacktest(backtester,pricesTT)
```

```
backtester = 
  backtestEngine with properties:

               Strategies: [1x1 backtestStrategy]
             RiskFreeRate: 0
           CashBorrowRate: 0
          RatesConvention: "Annualized"
                    Basis: 0
    InitialPortfolioValue: 10000
           DateAdjustment: "Previous"
                NumAssets: 7
                  Returns: [250x1 timetable]
                Positions: [1x1 struct]
                 Turnover: [250x1 timetable]
                  BuyCost: [250x1 timetable]
                 SellCost: [250x1 timetable]
```

```
                        Fees: [1x1 struct]
```

Backtest Summary

Use the `summary` function to generate a summary table of backtest results.

```
% Examing results. The summary table shows several performance metrics.
summary(backtester)
```

```
ans=9×1 table
                       EqualWeighted
                       _____________

    TotalReturn            0.22943  
    SharpeRatio            0.11415  
    Volatility           0.0075013  
    AverageTurnover     0.00054232  
    MaxTurnover           0.038694  
    AverageReturn       0.00085456  
    MaxDrawdown           0.098905  
    AverageBuyCost        0.030193  
    AverageSellCost       0.030193  
```

Warm Starting Backtests

When running a backtest in MATLAB®, you need to understand what the initial conditions are when the backtest begins. The initial weights for each strategy, the size of the strategy lookback window, and any potential split of the dataset into training and testing partitions affects the results of the backtest. This example shows how to use the `runBacktest` function with the `'Start'` and `'End'` name-value pair arguments that interact with the `'LookbackWindow'` and `'RebalanceFrequency'` properties of the `backtestStrategy` object to "warm start" a backtest.

Load Data

Load one year of stock price data. For readability, this example uses only a subset of the DJIA stocks.

```
% Read table of daily adjusted close prices for 2006 DJIA stocks.
T = readtable('dowPortfolio.xlsx');

% Prune the table to include only the dates and selected stocks.
timeColumn = "Dates";
assetSymbols = ["BA", "CAT", "DIS", "GE", "IBM", "MCD", "MSFT"];
T = T(:,[timeColumn assetSymbols]);

% Convert to timetable.
pricesTT = table2timetable(T,'RowTimes','Dates');

% View the final asset price timetable.
head(pricesTT)
```

```
       Dates        BA       CAT      DIS      GE      IBM      MCD     MSFT 
    ___________    _____    _____    _____    ____    _____    _____    _____

    03-Jan-2006    68.63    55.86    24.18    33.6    80.13    32.72    26.19
    04-Jan-2006    69.34    57.29    23.77    33.56    80.03    33.01    26.32
    05-Jan-2006    68.53    57.29    24.19    33.47    80.56    33.05    26.34
    06-Jan-2006    67.57    58.43    24.52     33.7    82.96    33.25    26.26
    09-Jan-2006    67.01    59.49    24.78    33.61    81.76    33.88    26.21
    10-Jan-2006    67.33    59.25    25.09    33.43     82.1    33.91    26.35
    11-Jan-2006     68.3    59.28    25.33    33.66    82.19     34.5    26.63
    12-Jan-2006     67.9    60.13    25.41    33.25    81.61    33.96    26.48
```

Create Strategy

This example backtests an "inverse variance" strategy. The inverse variance rebalance function is implemeted in the Local Functions on page 15-255 section. For more information on creating backtest strategies, see `backtestStrategy`. The inverse variance strategy uses the covariance of asset returns to make decisions about asset allocation. The `LookbackWindow` for this strategy must contain at least 30 days of trailing data (about 6 weeks), and at most, 60 days (about 12 weeks).

Set `RebalanceFrequency` for `backtestStrategy` to rebalance the portfolio every 25 days.

```
% Create the strategy
minLookback = 30;
maxLookback = 60;
ivStrategy = backtestStrategy("InverseVariance",@inverseVarianceFcn, ...
    'RebalanceFrequency',25, ...
    'LookbackWindow',[minLookback maxLookback], ...
    'TransactionCosts',[0.0025 0.005])
```

```
ivStrategy = 
  backtestStrategy with properties:

                      Name: "InverseVariance"
              RebalanceFcn: @inverseVarianceFcn
        RebalanceFrequency: 25
          TransactionCosts: [0.0025 0.0050]
            LookbackWindow: [30 60]
            InitialWeights: [1x0 double]
             ManagementFee: 0
     ManagementFeeSchedule: 1y
            PerformanceFee: 0
    PerformanceFeeSchedule: 1y
         PerformanceHurdle: 0
                  UserData: [0x0 struct]
            EngineDataList: [0x0 string]
```

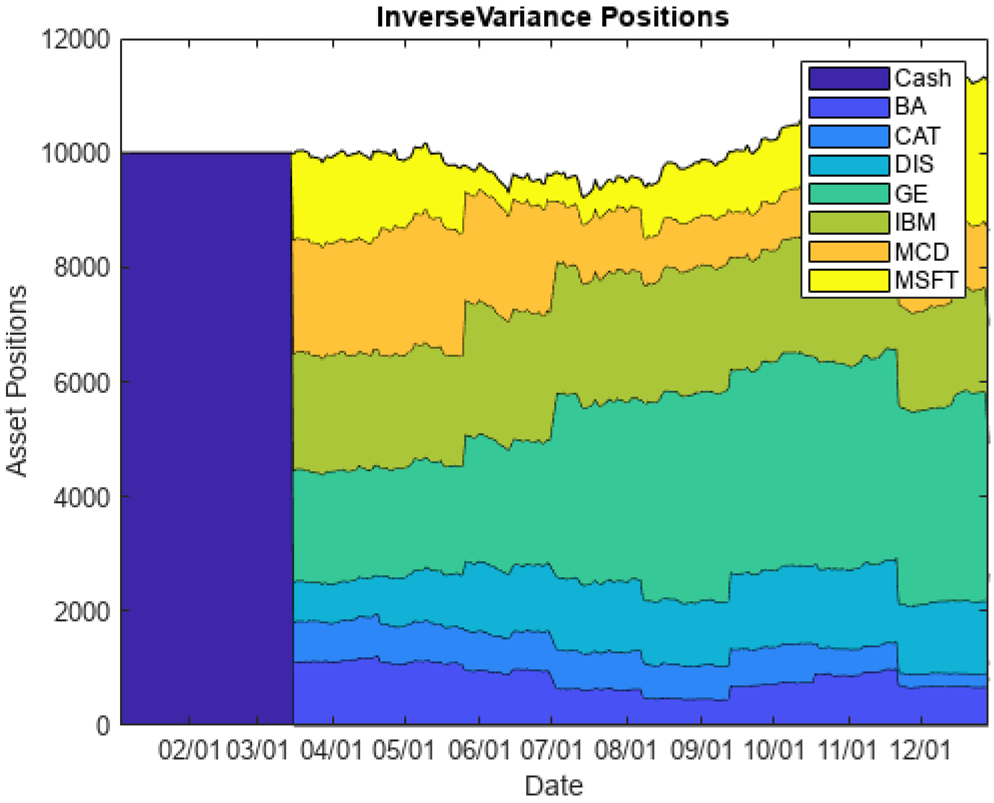
Run Backtest and Examine Results

Create a backtesting engine and run a backtest over a year of stock data. For more information on creating backtest engines, see `backtestEngine`.

```
% Create the backtest engine.
backtester = backtestEngine(ivStrategy);
% Run the backtest.
backtester = runBacktest(backtester,pricesTT);
```

Use the `assetAreaPlot` helper function, defined in the Local Functions on page 15-255 section of this example, to display the change in the asset allocation over the course of the backtest.

```
assetAreaPlot(backtester,"InverseVariance")
```



Notice that the inverse variance strategy begins all in cash and remains in that state for about 2.5 months. This is because the `backtestStrategy` object does not have a specified set of initial weights, which you specify using the `InitialPortfolioValue` name-value pair argument. The inverse variance strategy requires 30 days of trailing asset price history before rebalancing. You can use the `printRebalanceTable` helper function, defined in the Local Functions on page 15-255 section, to display the rebalance schedule.

```
printRebalanceTable(ivStrategy,pricesTT,minLookback);
```

```
    First Day of Data    Backtest Start Date    Minimum Days to Rebalance
    _________________    ___________________    _________________________

       03-Jan-2006           03-Jan-2006                   30            


    Rebalance Dates    Days of Available Price History    Enough Data to Rebalance
    _______________    _______________________________    ________________________

      08-Feb-2006                     26                            "No"          
      16-Mar-2006                     51                            "Yes"         
      21-Apr-2006                     76                            "Yes"         
      26-May-2006                    101                            "Yes"         
      03-Jul-2006                    126                            "Yes"         
      08-Aug-2006                    151                            "Yes"         
      13-Sep-2006                    176                            "Yes"         
      18-Oct-2006                    201                            "Yes"         
      22-Nov-2006                      226                             "Yes"  
      29-Dec-2006                      251                             "Yes"  
```

The first rebalance date comes on February 8 but the strategy does not have enough price history to fill out a valid lookback window (minimum is 30 days), so no rebalance occurs. The next rebalance date is on March 16, a full 50 days into the backtest.

This situation is not ideal as these 50 days sitting in an all-cash position represent approximately 20% of the total backtest. Consequently, when the backtesting engine reports on the performance of the strategy (that is, the total return, Sharpe ratio, volatility, and so on), the results do not reflect the "true" strategy performance because the strategy only began to make asset allocation decisions only about 20% into the backtest.

Warm Start Backtest

It is possible to "warm start" the backtest. A warm start means that the backtest results reflect the strategy performance in the market conditions reflected in the price timetable. To start, set the initial weights of the strategy to avoid starting all in cash.

The inverse variance strategy requires 30 days of price history to fill out a valid lookback window, so you can partition the price data set into two sections, a "warm-up" set and a "test" set.

```
warmupRange = 1:30;
% The 30th row is included in both ranges since the day 30 price is used
% to compute the day 31 returns.
testRange = 30:height(pricesTT);
```

Use the warm-up partition to set the initial weights of the inverse variance strategy. By doing so, you can begin the backtest with the strategy already "running" and avoid the initial weeks spent in the cash position.

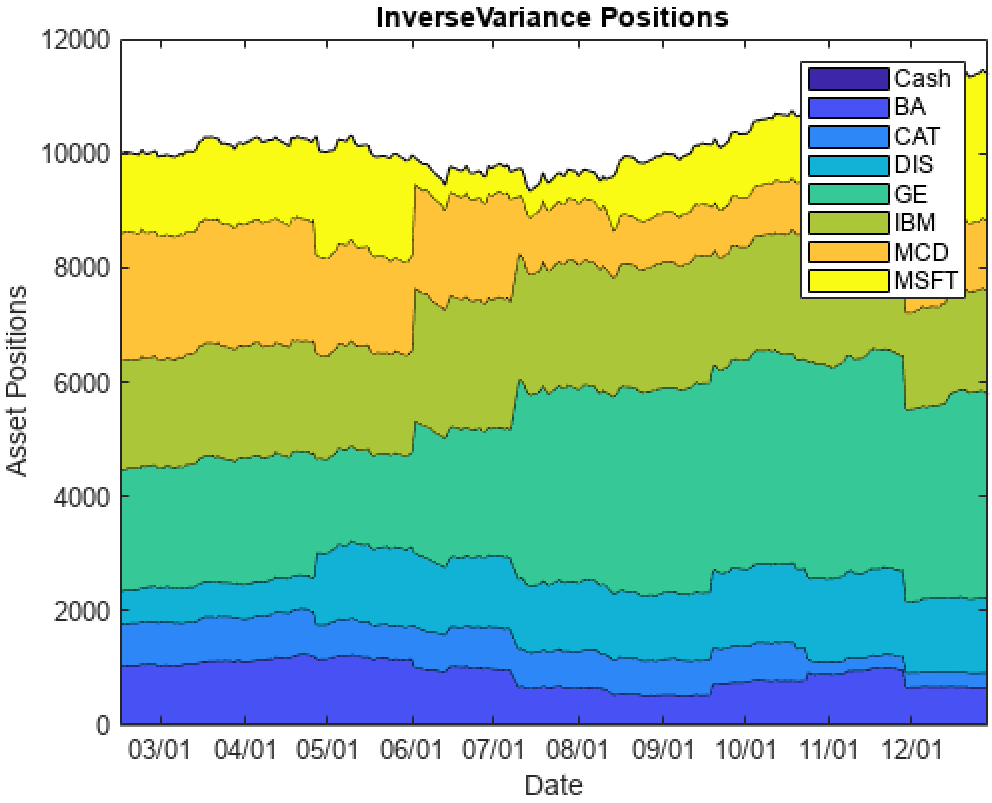
```
% Use the rebalance function to set the initial weights. This might
% or might not be possible for other strategies depending on the details of
% the strategy logic.
initWeights = inverseVarianceFcn([],pricesTT(warmupRange,:));
```

Update the strategy and rerun the backtest. Since the warm-up range is used to initialize the inverse variance strategy, you must omit this data from the backtest to avoid a look-ahead bias, or "seeing the future," and to backtest only over the "test range."

```
% Set the initial weights on the strategy in the backtester. You can do this when you
% create the strategy as well, using the 'InitialWeights' parameter.
backtester.Strategies(1).InitialWeights = initWeights;

% Rerun the backtest over the "test" range.
backtester = runBacktest(backtester,pricesTT(testRange,:));
```

When you generate the area plot, you can see that the issue where the strategy is in cash for the first portion of the backtest is avoided.

```
assetAreaPlot(backtester,"InverseVariance")
```



However, if you look at the rebalance table, you can see that the strategy still "missed" the first rebalance date. When you run the backtest over the test range of the data set, the first rebalance date is on March 22. This is because the warm-up range is omitted from the price history and the strategy had only 26 days of history available on that date (less than the minimum 30 days required for the lookback window). Therefore, the March 22 rebalance is skipped.

To avoid backtesting over the warm-up range, the range was removed it from the data set. This means the new backtest start date and all subsequent rebalance dates are 30 days later. The price history data contained in the warm-up range was completely removed, so when the backtest engine hit the first rebalance date the price history was insufficient to rebalance.

```
printRebalanceTable(ivStrategy,pricesTT(testRange,:),minLookback);
```

```
    First Day of Data    Backtest Start Date    Minimum Days to Rebalance
    _________________    ___________________    _________________________

       14-Feb-2006           14-Feb-2006                   30            


    Rebalance Dates    Days of Available Price History    Enough Data to Rebalance
    _______________    _______________________________    ________________________

      22-Mar-2006                     26                            "No"          
      27-Apr-2006                     51                            "Yes"         
      02-Jun-2006                     76                            "Yes"         
      10-Jul-2006                    101                            "Yes"         
      14-Aug-2006                     126                             "Yes"
      19-Sep-2006                     151                             "Yes"
      24-Oct-2006                     176                             "Yes"
      29-Nov-2006                     201                             "Yes"
```

This scenario is also not correct since the original price timetable (warm-up and test partitions together) does have enough price history by March 22 to fill out a valid lookback window. However, the earlier data is not available to the backtest engine because the backtest was run using only the test partition.

Use `Start` and End Parameters for `runBacktest`

The ideal workflow in this situation is to both *omit* the warm-up data range from the backtest to avoid the look-ahead bias but *include* the warm-up data in the price history to be able to fill out the lookback window of the strategy with all available price history data. You can do so by using the `'Start'` parameter for the `runBacktest` function.

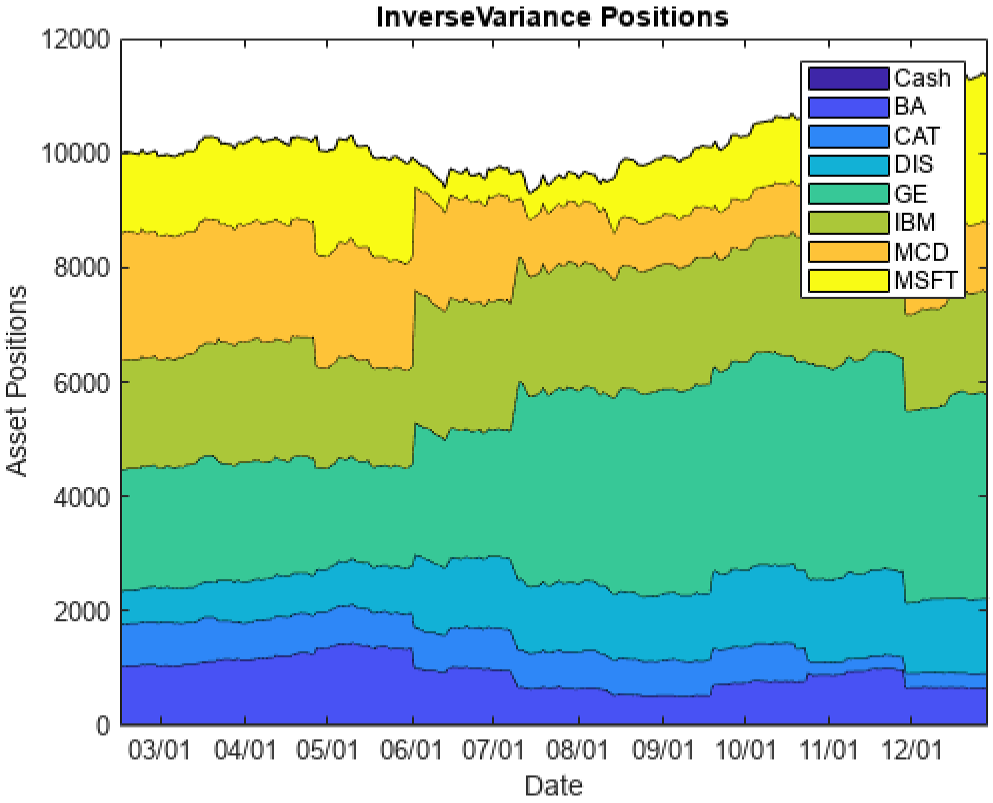
The `'Start'` and `'End'` name-value pair arguments for `runBacktest` enable you to start and end the backtest on specific dates. You can specify `'Start'` and `'End'` as rows of the prices timetable or as datetime values (see the documentation for the `runBacktest` function for details). The `'Start'` argument lets the backtest begin on a particular date while giving the backtest engine access to the full data set.

Rerun the backtest using the `'Start'` name-value pair argument rather than only running on a partition of the original data set.

```
% Rerun the backtest starting on the last day of the warmup range.
startRow = warmupRange(end);
backtester = runBacktest(backtester,pricesTT,'Start',startRow);
```

Plot the new asset area plot.

```
assetAreaPlot(backtester,"InverseVariance")
```



View the new rebalance table with the new `'Start'` parameter.

```
printRebalanceTable(ivStrategy,pricesTT,minLookback,startRow);
```

```
    First Day of Data    Backtest Start Date    Minimum Days to Rebalance
    _________________    ___________________    _________________________

       03-Jan-2006           14-Feb-2006                   30            


    Rebalance Dates    Days of Available Price History    Enough Data to Rebalance
    _______________    _______________________________    ________________________

      22-Mar-2006                     55                            "Yes"         
      27-Apr-2006                     80                            "Yes"         
      02-Jun-2006                    105                            "Yes"         
      10-Jul-2006                    130                            "Yes"         
      14-Aug-2006                    155                            "Yes"         
      19-Sep-2006                    180                            "Yes"         
      24-Oct-2006                    205                            "Yes"         
      29-Nov-2006                    230                            "Yes"         
```

The inverse variance strategy now has enough data to rebalance on the first rebalance date (March 22) and the backtest is "warm started." By using the original data set, the first day of data remains January 3, and the `'Start'` parameter allows you to move the backtest start date forward to avoid the warm-up range.

Even though the results are not dramatically different, this example illustrates the interaction between the LookbackWindow and RebalanceFrequency name-value pair arguments for a backtestStrategy object and the range of data used in the runBacktest when you evaluate the performance of a strategy in a backtest.

Local Functions

The strategy rebalance function is implemented as follows. For more information on creating strategies and writing rebalance functions, see backtestStrategy.

```
function new_weights = inverseVarianceFcn(current_weights, pricesTT)
% Inverse-variance portfolio allocation.

assetReturns = tick2ret(pricesTT);
assetCov = cov(assetReturns{:,:});
new_weights = 1 ./ diag(assetCov);
new_weights = new_weights / sum(new_weights);

end
```

This helper function plots the asset allocation as an area plot.

```
function assetAreaPlot(backtester,strategyName)

t = backtester.Positions.(strategyName).Time;
positions = backtester.Positions.(strategyName).Variables;
h = area(t,positions);
title(sprintf('%s Positions',strategyName));
xlabel('Date');
ylabel('Asset Positions');
datetick('x','mm/dd','keepticks');
xlim([t(1) t(end)])
oldylim = ylim;
ylim([0 oldylim(2)]);
cm = parula(numel(h));
for i = 1:numel(h)
    set(h(i),'FaceColor',cm(i,:));
end
legend(backtester.Positions.(strategyName).Properties.VariableNames)

end
```

This helper function generates a table of rebalance dates along with the available price history at each date.

```
function printRebalanceTable(strategy,pricesTT,minLookback,startRow)

if nargin < 4
    startRow = 1;
end

allDates = pricesTT.(pricesTT.Properties.DimensionNames{1});
rebalanceDates = allDates(startRow:strategy.RebalanceFrequency:end);
[~,rebalanceIndices] = ismember(rebalanceDates,pricesTT.Dates);

disp(table(allDates(1),rebalanceDates(1),minLookback,'VariableNames',{'First Day of Data','Backte
fprintf('\n\n');
numHistory = rebalanceIndices(2:end);
```

```
sufficient = repmat("No",size(numHistory));
sufficient(numHistory > minLookback) = "Yes";
disp(table(rebalanceDates(2:end),rebalanceIndices(2:end),sufficient,'VariableNames',{'Rebalance

end
```

Input Arguments

backtester — Backtesting engine

backtestEngine object

Backtesting engine, specified as a backtestEngine object. Use backtestEngine to create the backtester object.

Data Types: object

pricesTT — Asset prices

timetable

Asset prices, specified as a timetable of asset prices that the backtestEngine uses to backtest the strategies. Each column of the prices timetable must contain a timeseries of prices for an asset. Historical asset prices must be adjusted for splits and dividends.

Data Types: timetable

signalTT — Signal data

timetable

(Optional) Signal data, specified as a timetable of trading signals that the strategies use to make trading decisions. signalTT is optional. If provided, the backtestEngine calls the strategy rebalance functions with both asset price data and signal data. The signalTT timetable must have the same time dimension as the pricesTT timetable.

Data Types: timetable

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: backtester = runBacktest(backtester,assetPrices,'Start',50,'End',100)

Start — Time step to start backtest

1 (default) | integer | datetime

Time step to start the backtest, specified as the comma-separated pair consisting of 'Start' and a scalar integer or datetime.

If an integer, the Start time refers to the row in the pricesTT timetable where the backtest begins.

If a datetime object, the backtest will begin at the first time in the prices timetable that occurs on or after the 'Start' parameter. The backtest will end on the last time in the pricesTT timetable that occurs on or before the 'End' parameter. The 'Start' and 'End' parameters set the boundary of the data that is included in the backtest.

Data Types: double | datetime

End — Time step to end backtest
Last row in pricesTT timetable (default) | integer | datetime

Time step to end the backtest, specified as the comma-separated pair consisting of 'End' and a scalar integer or datetime.

If an integer, the End time refers to the row in the pricesTT timetable where the backtest ends.

If a datetime object, the backtest will end on the last time in the pricesTT timetable that occurs on or before the 'End' parameter.

Data Types: double | datetime

Output Arguments

backtester — Backtesting engine
backtestEngine object

Backtesting engine, returned as an updated backtestEngine object. After backtesting is complete, runBacktest populates several properties in the backtestEngine object with the results of the backtest. You can summarize the results by using the summary function.

Version History

Introduced in R2020b

See Also

backtestStrategy | backtestEngine | summary | equityCurve

Topics

“Backtest Investment Strategies Using Financial Toolbox™” on page 4-231
“Backtest Investment Strategies with Trading Signals” on page 4-244
“Backtest with Brinson Attribution to Evaluate Portfolio Performance” on page 4-311

summary

Generate summary table of backtest results

Syntax

```
summaryTable = summary(backtester)
```

Description

`summaryTable = summary(backtester)` generates a table of metrics to summarize the backtest. Each row of the table is a calculated metric and each column represents a strategy. You must run the summary function only after running the `runBacktest` function.

Examples

Examining Backtest Results

The MATLAB® backtesting engine runs backtests of portfolio investment strategies over time series of asset price data. You can use `summary` to compare multiple strategies over the same market scenario. This example shows how to examine the results of a backtest with two strategies.

Load Data

Load one year of stock price data. For readability, this example uses a subset of the DJIA stocks.

```
% Read table of daily adjusted close prices for 2006 DJIA stocks
T = readtable('dowPortfolio.xlsx');

% Prune the table to include only the dates and selected stocks
timeColumn = "Dates";
assetSymbols = ["BA", "CAT", "DIS", "GE", "IBM", "MCD", "MSFT"];
T = T(:,[timeColumn assetSymbols]);

% Convert to timetable
pricesTT = table2timetable(T,'RowTimes','Dates');

% View the final asset price timetable
head(pricesTT)
```

```
       Dates         BA       CAT      DIS       GE       IBM      MCD      MSFT 
    ___________    _____    _____    _____    _____    _____    _____    _____

    03-Jan-2006    68.63    55.86    24.18     33.6    80.13    32.72    26.19
    04-Jan-2006    69.34    57.29    23.77    33.56    80.03    33.01    26.32
    05-Jan-2006    68.53    57.29    24.19    33.47    80.56    33.05    26.34
    06-Jan-2006    67.57    58.43    24.52     33.7    82.96    33.25    26.26
    09-Jan-2006    67.01    59.49    24.78    33.61    81.76    33.88    26.21
    10-Jan-2006    67.33    59.25    25.09    33.43     82.1    33.91    26.35
    11-Jan-2006     68.3    59.28    25.33    33.66    82.19     34.5    26.63
    12-Jan-2006     67.9    60.13    25.41    33.25    81.61    33.96    26.48
```

The inverse variance strategy requires some price history to initialize, so you can allocate a portion of the data to use for setting initial weights. By doing this, you can "warm start" the backtest.

```
warmupRange = 1:20;
testRange = 21:height(pricesTT);
```

Create Strategies

Define an investment strategy by using the `backtestStrategy` function. This example builds two strategies:

- Equal weighted
- Inverse variance

This example does not provide details on how to build the strategies. For more information on creating strategies, see `backtestStrategy`. The strategy rebalance functions are implemented in the Rebalance Functions on page 15-268 section.

```
% Create the strategies
ewInitialWeights = equalWeightFcn([],pricesTT(warmupRange,:));
ewStrategy = backtestStrategy("EqualWeighted",@equalWeightFcn, ...
    'RebalanceFrequency',20, ...
    'TransactionCosts',[0.0025 0.005], ...
    'LookbackWindow',0, ...
    'InitialWeights',ewInitialWeights)
```

```
ewStrategy = 
  backtestStrategy with properties:

                      Name: "EqualWeighted"
              RebalanceFcn: @equalWeightFcn
        RebalanceFrequency: 20
          TransactionCosts: [0.0025 0.0050]
            LookbackWindow: 0
            InitialWeights: [0.1429 0.1429 0.1429 0.1429 0.1429 0.1429 0.1429]
             ManagementFee: 0
     ManagementFeeSchedule: 1y
            PerformanceFee: 0
    PerformanceFeeSchedule: 1y
         PerformanceHurdle: 0
                  UserData: [0x0 struct]
            EngineDataList: [0x0 string]
```

```
ivInitialWeights = inverseVarianceFcn([],pricesTT(warmupRange,:));
ivStrategy = backtestStrategy("InverseVariance",@inverseVarianceFcn, ...
    'RebalanceFrequency',20, ...
    'TransactionCosts',[0.0025 0.005], ...
    'InitialWeights',ivInitialWeights)
```

```
ivStrategy = 
  backtestStrategy with properties:

                      Name: "InverseVariance"
              RebalanceFcn: @inverseVarianceFcn
        RebalanceFrequency: 20
          TransactionCosts: [0.0025 0.0050]
```

```
            LookbackWindow: [0 Inf]
            InitialWeights: [0.1401 0.0682 0.0795 0.2187 0.1900 0.1875 0.1160]
             ManagementFee: 0
     ManagementFeeSchedule: 1y
            PerformanceFee: 0
    PerformanceFeeSchedule: 1y
         PerformanceHurdle: 0
                  UserData: [0x0 struct]
            EngineDataList: [0x0 string]
```

```
% Aggregate the strategies into an array
strategies = [ewStrategy ivStrategy];
```

Run Backtest

Create a backtesting engine and run a backtest over a year of stock data. For more information on creating backtesting engines, see `backtestEngine`. The software initializes several properties of the `backtestEngine` object to empty. These read-only properties are populated by the engine after you run the backtest.

```
% Create the backtesting engine using the default settings
backtester = backtestEngine(strategies)
```

```
backtester = 
  backtestEngine with properties:

               Strategies: [1x2 backtestStrategy]
             RiskFreeRate: 0
           CashBorrowRate: 0
          RatesConvention: "Annualized"
                    Basis: 0
    InitialPortfolioValue: 10000
           DateAdjustment: "Previous"
                NumAssets: []
                  Returns: []
                Positions: []
                 Turnover: []
                  BuyCost: []
                 SellCost: []
                     Fees: []
```

Run the backtest using `runBacktest`.

```
% Run the backtest
backtester = runBacktest(backtester,pricesTT(testRange,:));
```

Examine Summary Results

The `summary` function uses the results of the backtest and returns a table of high-level results from the backtest.

```
s1 = summary(backtester)
```

```
s1=9×2 table
                       EqualWeighted    InverseVariance
                       _____________    _______________
```

```
    TotalReturn            0.17567            0.17155
    SharpeRatio           0.097946            0.10213
    Volatility           0.0074876          0.0069961
    AverageTurnover      0.0007014          0.0024246
    MaxTurnover           0.021107           0.097472
    AverageReturn       0.00073178         0.00071296
    MaxDrawdown           0.097647           0.096299
    AverageBuyCost        0.018532           0.061913
    AverageSellCost       0.037064            0.12383
```

Each row of the table output is a measurement of the performance of a strategy. Each strategy occupies a column. The `summary` function reports on the following metrics:

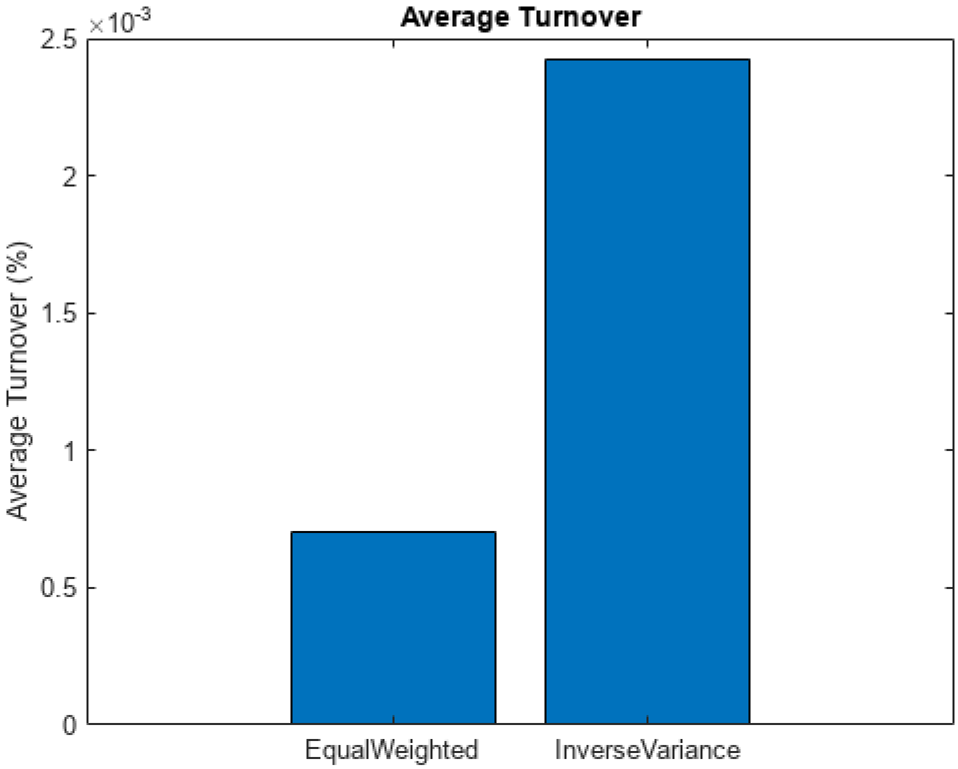
- `TotalReturn` — The nonannulaized total return of the strategy, inclusive of fees, over the full backtest period.
- `SharpeRatio` — The nonannualized Sharpe ratio of each strategy over the backtest. For more information, see `sharpe`.
- `Volatility` — The nonannualized standard deviation of per-time-step strategy returns.
- `AverageTurnover` — The average per-time-step portfolio turnover, expressed as a decimal percentage.
- `MaxTurnover` — The maximum portfolio turnover in a single rebalance, expressed as a decimal percentage.
- `AverageReturn` —The arithmetic mean of the per-time step portfolio returns.
- `MaxDrawdown` — The maximum drawdown of the portfolio, expressed as a decimal percentage. For more information, see `maxdrawdown`.
- `AverageBuyCost` — The average per-time-step transaction costs the portfolio incurred for asset purchases.
- `AverageSellCost` — The average per-time-step transaction costs the portfolio incurred for asset sales.

Sometimes it is useful to transpose the `summary` table when plotting the metrics of different strategies.

```
s2 = rows2vars(s1);
s2.Properties.VariableNames{1} = 'StrategyName'
```

```
s2=2×10 table
       StrategyName        TotalReturn    SharpeRatio    Volatility    AverageTurnover    MaxTurn
    ___________________    ___________    ___________    __________    _______________    _______

    {'EqualWeighted'  }      0.17567       0.097946      0.0074876        0.0007014        0.0211
    {'InverseVariance'}      0.17155        0.10213      0.0069961        0.0024246        0.0974
```

```
bar(s2.AverageTurnover)
title('Average Turnover')
ylabel('Average Turnover (%)')
set(gca,'xticklabel',s2.StrategyName)
```



Examine Detailed Results

After you run the backtest, the `backtestEngine` object updates the read-only fields with the detailed results of the backtest. The `Returns`, `Positions`, `Turnover`, `BuyCost`, `SellCost`, and `Fees` properties each contain a timetable of results. Since this example uses daily price data in the backtest, these timetables hold daily results.

```
backtester
```

```
backtester = 
  backtestEngine with properties:

               Strategies: [1x2 backtestStrategy]
             RiskFreeRate: 0
           CashBorrowRate: 0
          RatesConvention: "Annualized"
                    Basis: 0
    InitialPortfolioValue: 10000
           DateAdjustment: "Previous"
                NumAssets: 7
                  Returns: [230x2 timetable]
                Positions: [1x1 struct]
                 Turnover: [230x2 timetable]
                  BuyCost: [230x2 timetable]
                 SellCost: [230x2 timetable]
                     Fees: [1x1 struct]
```

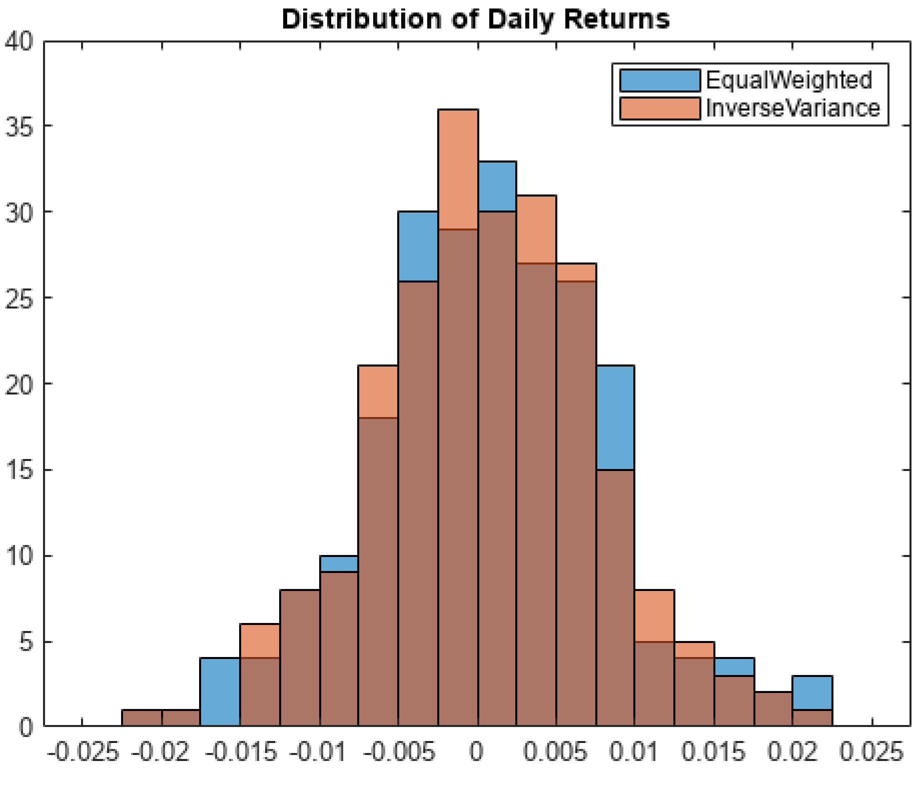
Returns

The `Returns` property holds a timetable of strategy (simple) returns for each time step. These returns are inclusive of all transaction fees.

```
head(backtester.Returns)
```

```
       Time         EqualWeighted    InverseVariance
    ___________    _____________    _______________

    02-Feb-2006      -0.007553        -0.0070957
    03-Feb-2006     -0.0037771         -0.003327
    06-Feb-2006     -0.0010094        -0.0014312
    07-Feb-2006      0.0053284         0.0020578
    08-Feb-2006      0.0099755         0.0095781
    09-Feb-2006     -0.0026871        -0.0014999
    10-Feb-2006      0.0048374         0.0059589
    13-Feb-2006     -0.0056868        -0.0051232
```

```
binedges = -0.025:0.0025:0.025;
h1 = histogram(backtester.Returns.EqualWeighted,'BinEdges',binedges);
hold on
histogram(backtester.Returns.InverseVariance,'BinEdges',binedges);
hold off
title('Distribution of Daily Returns')
legend([strategies.Name]);
```



Positions

The `Positions` property holds a structure of timetables, one per strategy.

```
backtester.Positions
```

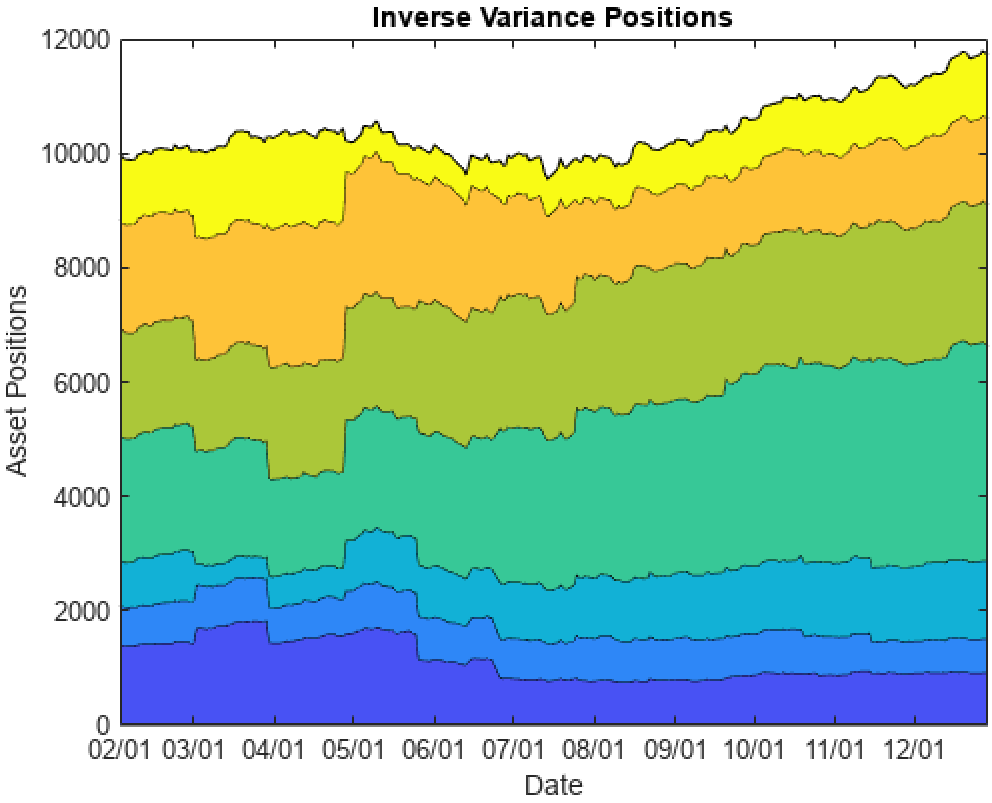
```
ans = struct with fields:
      EqualWeighted: [231x8 timetable]
    InverseVariance: [231x8 timetable]
```

The `Positions` timetable of each strategy holds the per-time-step positions for each asset as well as the `Cash` asset (which earns the risk-free rate). The `Positions` timetables contain one more row than the other results timetables because the `Positions` timetables include initial positions of the strategy as their first row. You can consider the initial positions as the `Time = 0` portfolio positions. In this example, the `Positions` timetables start with February 1, while the others start on February 2.

```
head(backtester.Positions.InverseVariance)
```

```
       Time              Cash         BA       CAT       DIS        GE       IBM       MCD       MS
    ___________    __________    ______    ______    ______    ______    ______    ______    ___

    01-Feb-2006             0    1401.2    682.17    795.14    2186.8    1900.1    1874.9    1159
    02-Feb-2006             0    1402.8    673.74    789.74    2170.8    1883.5    1863.6      11
    03-Feb-2006             0    1386.5     671.2     787.2    2167.3    1854.3    1890.5      11
    06-Feb-2006    3.2913e-12    1391.9    676.78    785.62    2161.1    1843.6    1899.1    1123
    07-Feb-2006    1.0994e-12      1400    661.66    840.23    2131.9    1851.6    1902.3    1114
    08-Feb-2006    2.2198e-12    1409.8     677.9    846.58    2160.4    1878.2      1911    1113
    09-Feb-2006    2.2165e-12    1414.8    674.35    840.87    2172.2      1869    1908.3    1102
    10-Feb-2006    1.1148e-12    1425.1    677.29     839.6    2195.8    1890.6    1909.3    1103
```

```
% Plot the change of asset allocation over time
t = backtester.Positions.InverseVariance.Time;
positions = backtester.Positions.InverseVariance.Variables;
h = area(t,positions);
title('Inverse Variance Positions');
xlabel('Date');
ylabel('Asset Positions');
datetick('x','mm/dd','keepticks');
ylim([0 12000])
xlim([t(1) t(end)])
cm = parula(numel(h));
for i = 1:numel(h)
    set(h(i),'FaceColor',cm(i,:));
end
```



Turnover

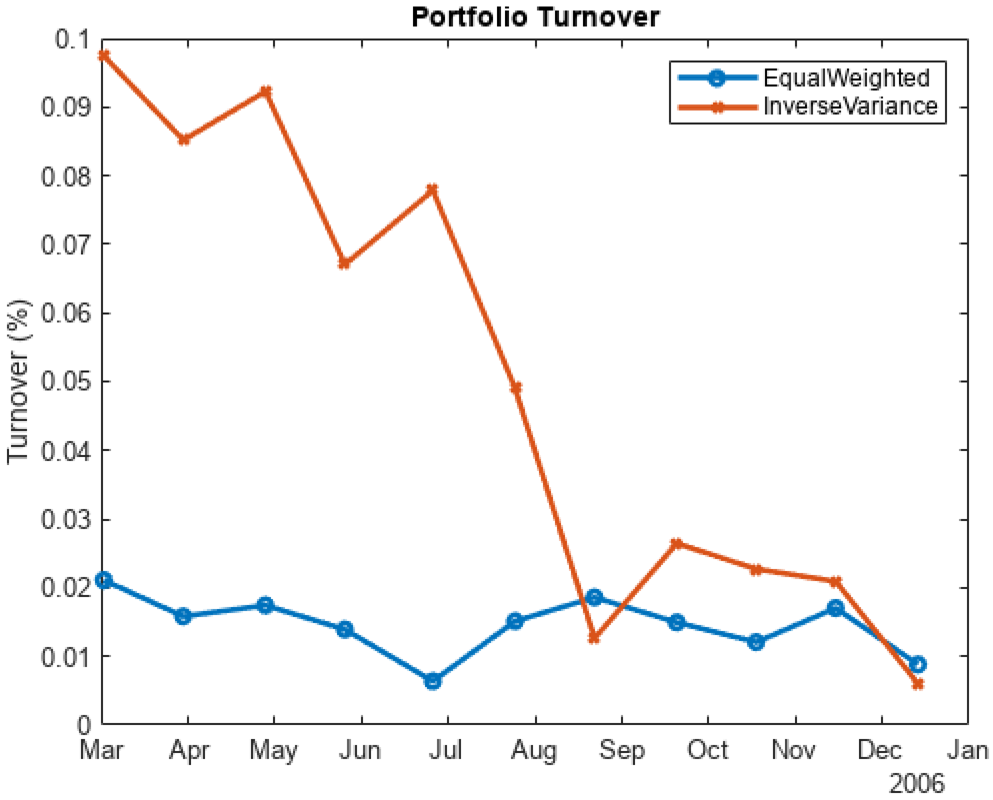
The `Turnover` timetable holds the per-time-step portfolio turnover.

```
head(backtester.Turnover)
```

```
       Time        EqualWeighted    InverseVariance
    ___________    _____________    _______________

    02-Feb-2006          0                 0       
    03-Feb-2006          0                 0       
    06-Feb-2006          0                 0       
    07-Feb-2006          0                 0       
    08-Feb-2006          0                 0       
    09-Feb-2006          0                 0       
    10-Feb-2006          0                 0       
    13-Feb-2006          0                 0       
```

Depending on your rebalance frequency, the `Turnover` table can contain mostly zeros. Removing these zeros when you visualize the portfolio turnover is useful.

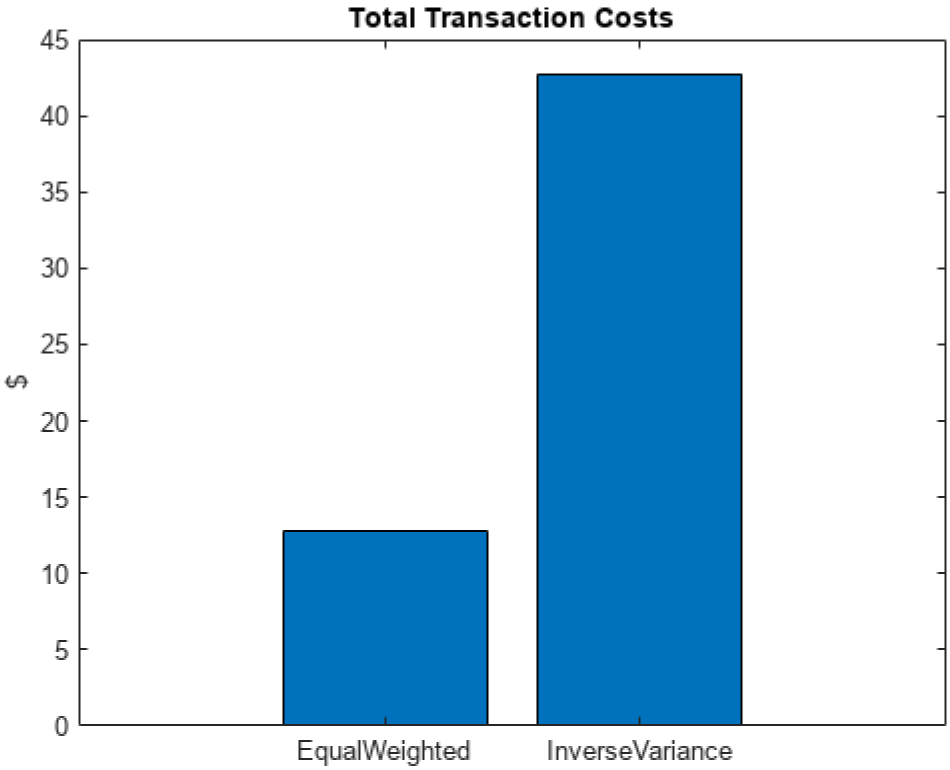
```
nonZeroIdx = sum(backtester.Turnover.Variables,2) > 0;
to = backtester.Turnover(nonZeroIdx,:);
plot(to.Time,to.EqualWeighted,'-o',to.Time,to.InverseVariance,'-x',...
    'LineWidth',2,'MarkerSize',5);
legend([strategies.Name]);
title('Portfolio Turnover');
ylabel('Turnover (%)');
```



BuyCost and SellCost

The BuyCost and SellCost timetables hold the per-time-step transaction fees for each type of transaction, purchases, and sales.

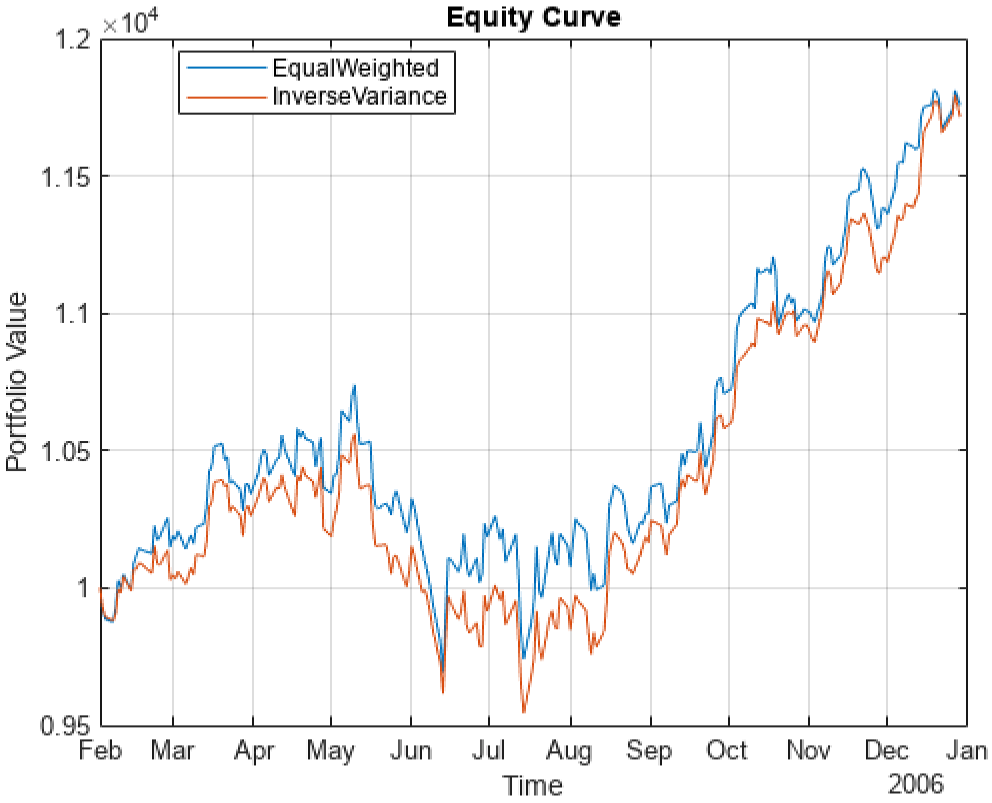
```
totalCost = sum(backtester.BuyCost{:,:}) + sum(backtester.SellCost{:,:});
bar(totalCost);
title('Total Transaction Costs');
ylabel('$')
set(gca,'xticklabel',[strategies.Name])
```



Generate Equity Curve

Use `equityCurve` to plot the equity curve for the equal weighted and inverse variance strategies.

```
equityCurve(backtester)
```



Rebalance Functions

This section contains the implementation of the strategy rebalance functions. For more information on creating strategies and writing rebalance functions, see `backtestStrategy`.

```
function new_weights = equalWeightFcn(current_weights, pricesTT) %#ok<INUSL>
% Equal weighted portfolio allocation

nAssets = size(pricesTT, 2);
new_weights = ones(1,nAssets);
new_weights = new_weights / sum(new_weights);

end


function new_weights = inverseVarianceFcn(current_weights, pricesTT) %#ok<INUSL>
% Inverse-variance portfolio allocation

assetReturns = tick2ret(pricesTT);
assetCov = cov(assetReturns{:,:});
new_weights = 1 ./ diag(assetCov);
new_weights = new_weights / sum(new_weights);
```

```
end
```

Input Arguments

backtester — Backtesting engine
backtestEngine object

Backtesting engine, specified as a backtestEngine object. Use backtestEngine to create the backtesting engine and then use runBacktest to run a backtest.

Data Types: object

Output Arguments

summaryTable — Metrics summarizing backtest
table

Metrics summarizing the backtest, returned as a table where each row of the table is a calculated metric and each column represents a strategy. The reported metrics are as follows:

- TotalReturn — The total return of the strategy over the entire backtest
- SharpeRatio — The Sharpe ratio for each strategy
- Volatility — The volatility of each strategy over the backtest
- AverageTurnover — Average turnover per-time-step as a decimal percent
- MaxTurnover — Maximum turnover in a single time step
- AverageReturn — Average return per-time-step
- MaxDrawdown — Maximum portfolio drawdown as a decimal percent
- AverageBuyCost — Average per-time-step transaction costs for asset purchases
- AverageSellCost — Average per-time-step transaction costs for asset sales

Version History

Introduced in R2020b

See Also

backtestStrategy | backtestEngine | runBacktest | equityCurve

Topics

“Backtest Investment Strategies Using Financial Toolbox™” on page 4-231
“Backtest Investment Strategies with Trading Signals” on page 4-244
“Backtest with Brinson Attribution to Evaluate Portfolio Performance” on page 4-311

equityCurve

Plot equity curves of strategies

Syntax

```
equityCurve(backtester)
h = equityCurve(ax,backtester)
```

Description

`equityCurve(backtester)` plots the equity curves of each strategy that you create using `backtestStrategy`. After creating the backtesting engine using `backtestEngine` and running the backtest with `runBacktest`, use `equityCurve` to plot the strategies and compare their performance.

`h = equityCurve(ax,backtester)` additionally returns the figure handle `h`.

Examples

Generate Equity Curve for Backtest

The MATLAB® backtesting engine runs backtests of portfolio investment strategies over time series of asset price data. After creating a set of backtest strategies using `backtestStrategy` and the backtesting engine using `backtestEngine`, the `runBacktest` function executes the backtest. After using the `runBacktest` function to test investment strategies, you can run the `equityCurve` function to plot the equity curves of strategies.

Load Data

Load one year of stock price data. For readability, this example uses only a subset of the DJIA stocks.

```
% Read a table of daily adjusted close prices for 2006 DJIA stocks
T = readtable('dowPortfolio.xlsx');

% Prune the table to hold only the dates and selected stocks
timeColumn = "Dates";
assetSymbols = ["BA", "CAT", "DIS", "GE", "IBM", "MCD", "MSFT"];
T = T(:,[timeColumn assetSymbols]);

% Convert to timetable
pricesTT = table2timetable(T,'RowTimes','Dates');

% View the final asset price timetable
head(pricesTT)
```

```
       Dates        BA       CAT      DIS       GE       IBM      MCD      MSFT 
    ___________    _____    _____    _____    _____    _____    _____    _____

    03-Jan-2006    68.63    55.86    24.18     33.6    80.13    32.72    26.19
    04-Jan-2006    69.34    57.29    23.77    33.56    80.03    33.01    26.32
    05-Jan-2006    68.53    57.29    24.19    33.47    80.56    33.05    26.34
    06-Jan-2006    67.57    58.43    24.52     33.7    82.96    33.25    26.26
    09-Jan-2006    67.01    59.49    24.78    33.61    81.76    33.88    26.21
    10-Jan-2006    67.33    59.25    25.09    33.43     82.1    33.91    26.35
    11-Jan-2006     68.3    59.28    25.33    33.66    82.19     34.5    26.63
    12-Jan-2006     67.9    60.13    25.41    33.25    81.61    33.96    26.48
```

Create Strategy

Test an equal-weighted investment strategy. This strategy invests an equal portion of the available capital into each asset. This example does not describe how to create backtesting strategies. For more information on creating backtesting strategies, see `backtestStrategy`.

Set `'RebalanceFrequency'` to rebalance the portfolio every 60 days. This example does not use a lookback window to rebalance.

```
% Create the strategy
numAssets = size(pricesTT,2);
equalWeightsVector = ones(1,numAssets) / numAssets;
equalWeightsRebalanceFcn = @(~,~) equalWeightsVector;

ewStrategy = backtestStrategy("EqualWeighted",equalWeightsRebalanceFcn, ...
    'RebalanceFrequency',60, ...
    'LookbackWindow',0, ...
    'TransactionCosts',0.005, ...
    'InitialWeights',equalWeightsVector)
```

```
ewStrategy = 
  backtestStrategy with properties:

                      Name: "EqualWeighted"
              RebalanceFcn: @(~,~)equalWeightsVector
        RebalanceFrequency: 60
          TransactionCosts: 0.0050
            LookbackWindow: 0
            InitialWeights: [0.1429 0.1429 0.1429 0.1429 0.1429 0.1429 0.1429]
             ManagementFee: 0
     ManagementFeeSchedule: 1y
            PerformanceFee: 0
    PerformanceFeeSchedule: 1y
         PerformanceHurdle: 0
                  UserData: [0x0 struct]
            EngineDataList: [0x0 string]
```

Run Backtest

Create a backtesting engine and run a backtest over a year of stock data. For more information on creating backtest engines, see `backtestEngine`.

```
% Create the backtesting engine. The backtesting engine properties that hold the
% results are initialized to empty.
backtester = backtestEngine(ewStrategy)
```

```
backtester = 
  backtestEngine with properties:

               Strategies: [1x1 backtestStrategy]
```

```
             RiskFreeRate: 0
            CashBorrowRate: 0
           RatesConvention: "Annualized"
                     Basis: 0
     InitialPortfolioValue: 10000
            DateAdjustment: "Previous"
                 NumAssets: []
                   Returns: []
                 Positions: []
                  Turnover: []
                   BuyCost: []
                  SellCost: []
                      Fees: []
```

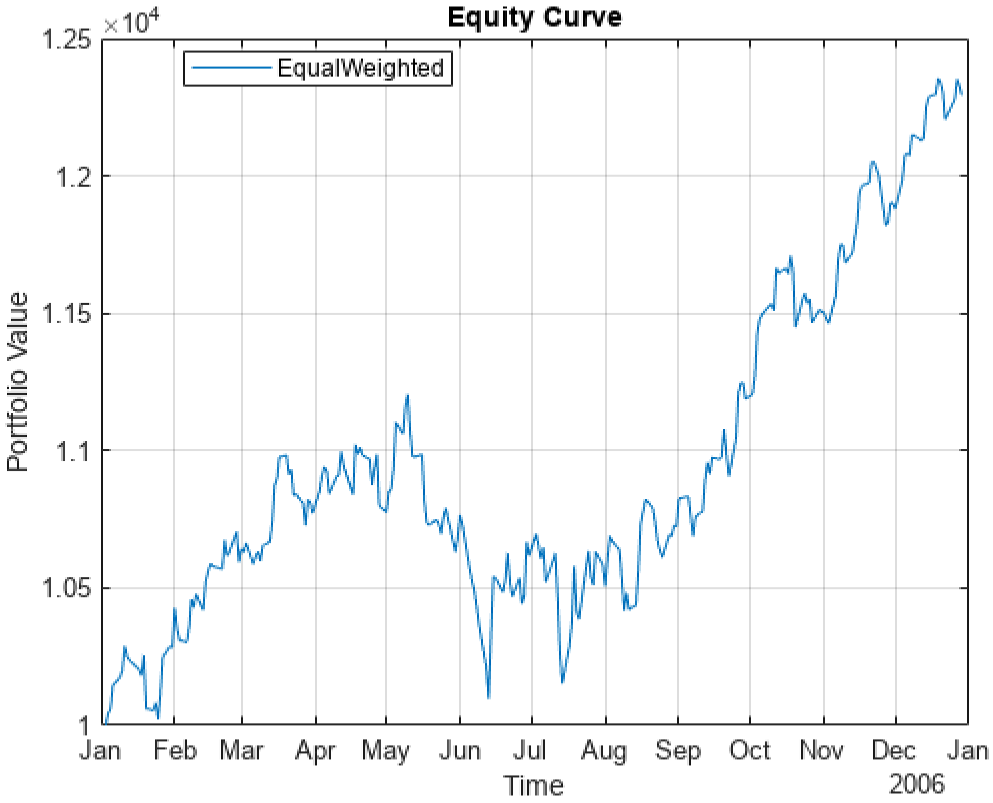
```
% Run the backtest. The empty properties are now populated with
% timetables of detailed backtest results.
backtester = runBacktest(backtester,pricesTT)
```

```
backtester = 
  backtestEngine with properties:

               Strategies: [1x1 backtestStrategy]
             RiskFreeRate: 0
            CashBorrowRate: 0
           RatesConvention: "Annualized"
                     Basis: 0
     InitialPortfolioValue: 10000
            DateAdjustment: "Previous"
                 NumAssets: 7
                   Returns: [250x1 timetable]
                 Positions: [1x1 struct]
                  Turnover: [250x1 timetable]
                   BuyCost: [250x1 timetable]
                  SellCost: [250x1 timetable]
                      Fees: [1x1 struct]
```

Generate Equity Curve

Use the `equityCurve` to plot the equity curve for the equal-weight strategy.

```
equityCurve(backtester)
```



Input Arguments

backtester — Backtesting engine

backtestEngine object

Backtesting engine, specified as a backtestEngine object. Use backtestEngine to create the backtester object.

Note The backtester argument must use a backtestEngine object that has been run using runBacktest.

Data Types: object

ax — Valid axis object

ax object

(Optional) Valid axis object, specified as an ax object that you create using axes. The plot is created on the axes specified by the optional ax argument instead of on the current axes (gca). The optional argument ax can precede any of the input argument combinations.

Data Types: object

Output Arguments

h — Figure handle
handle object

Figure handle for the equity curve plot, returned as handle object.

Version History

Introduced in R2021a

See Also

`backtestStrategy` | `backtestEngine` | `runBacktest` | `summary`

Topics

"Backtest Investment Strategies Using Financial Toolbox™" on page 4-231
"Backtest Investment Strategies with Trading Signals" on page 4-244
"Backtest with Brinson Attribution to Evaluate Portfolio Performance" on page 4-311

simByEuler

Simulate `Bates` sample paths by Euler approximation

Syntax

```
[Paths,Times,Z,N] = simByEuler(MDL,NPeriods)
[Paths,Times,Z,N] = simByEuler(___,Name,Value)
```

Description

`[Paths,Times,Z,N] = simByEuler(MDL,NPeriods)` simulates `NTrials` sample paths of Bates bivariate models driven by `NBrowns` Brownian motion sources of risk and `NJumps` compound Poisson processes representing the arrivals of important events over `NPeriods` consecutive observation periods. The simulation approximates continuous-time stochastic processes by the Euler approach.

`[Paths,Times,Z,N] = simByEuler(___,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

You can perform quasi-Monte Carlo simulations using the name-value arguments for `MonteCarloMethod`, `QuasiSequence`, and `BrownianMotionMethod`. For more information, see "Quasi-Monte Carlo Simulation" on page 14-62.

Examples

Simulate Bates Sample Paths by Euler Approximation

Create a `bates` object.

```
AssetPrice = 80;
            Return = 0.03;
            JumpMean = 0.02;
            JumpVol = 0.08;
            JumpFreq = 0.1;

            V0 = 0.04;
            Level = 0.05;
            Speed = 1.0;
            Volatility = 0.2;
            Rho = -0.7;
            StartState = [AssetPrice;V0];
            Correlation = [1 Rho;Rho 1];

batesObj = bates(Return, Speed, Level, Volatility,...
                JumpFreq, JumpMean, JumpVol,'startstate',StartState,...
                'correlation',Correlation)
```

```
batesObj = 
   Class BATES: Bates Bivariate Stochastic Volatility
   --------------------------------------------------
     Dimensions: State = 2, Brownian = 2
   --------------------------------------------------
      StartTime: 0
     StartState: 2x1 double array
    Correlation: 2x2 double array
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
         Return: 0.03
          Speed: 1
          Level: 0.05
     Volatility: 0.2
       JumpFreq: 0.1
       JumpMean: 0.02
        JumpVol: 0.08
```

Use `simByEuler` to simulate `NTrials` sample paths of this Bates bivariate model driven by `NBrowns` Brownian motion sources of risk and `NJumps` compound Poisson processes representing the arrivals of important events over `NPeriods` consecutive observation periods. This function approximates continuous-time stochastic processes by the Euler approach.

```
NPeriods = 2;
[Paths,Times,Z,N] = simByEuler(batesObj,NPeriods)
```

```
Paths = 3×2

   80.0000    0.0400
   90.8427    0.0873
   32.7458    0.1798


Times = 3×1

     0
     1
     2


Z = 2×2

    0.5377    0.9333
   -2.2588    2.1969


N = 2×1

     0
     0
```

The output `Paths` is `3 x 2` dimension. Row number `3` is from `NPeriods + 1`. The first row is defined by the `bates` name-value pair argument `StartState`. The remaining rows are simulated data. `Paths` always has two columns because the Bates model is a bivariate model. In this case, the first column is the simulated asset price and the second column is the simulated volatilities.

Quasi-Monte Carlo Simulation with `simByEuler` Using a Bates Model

Create a `bates` object.

```
AssetPrice = 80;
            Return = 0.03;
            JumpMean = 0.02;
            JumpVol = 0.08;
            JumpFreq = 0.1;

            V0 = 0.04;
            Level = 0.05;
            Speed = 1.0;
            Volatility = 0.2;
            Rho = -0.7;
            StartState = [AssetPrice;V0];
            Correlation = [1 Rho;Rho 1];

batesObj = bates(Return, Speed, Level, Volatility,...
                JumpFreq, JumpMean, JumpVol,'startstate',StartState,...
                'correlation',Correlation)
```

```
batesObj = 
   Class BATES: Bates Bivariate Stochastic Volatility
   --------------------------------------------------
     Dimensions: State = 2, Brownian = 2
   --------------------------------------------------
      StartTime: 0
     StartState: 2x1 double array
    Correlation: 2x2 double array
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
         Return: 0.03
          Speed: 1
          Level: 0.05
     Volatility: 0.2
       JumpFreq: 0.1
       JumpMean: 0.02
        JumpVol: 0.08
```

Define the quasi-Monte Carlo simulation using the optional name-value arguments for `'MonteCarloMethod'`,`'QuasiSequence'`, and `'BrownianMotionMethod'`.

```
[paths,time,z] = simByEuler(batesObj,10,'ntrials',4096,'montecarlomethod','quasi','quasisequence
```

Input Arguments

`MDL` — Stochastic differential equation model

object

Stochastic differential equation model, specified as a `bates` object. You can create a `bates` object using `bates`.

Data Types: `object`

NPeriods — Number of simulation periods

positive scalar integer

Number of simulation periods, specified as a positive scalar integer. The value of NPeriods determines the number of rows of the simulated output series.

Data Types: double

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: [Paths,Times,Z,N] = simByEuler(bates,NPeriods,'DeltaTimes',dt)

NTrials — Simulated trials (sample paths)

1 (single path of correlated state variables) (default) | positive scalar integer

Simulated trials (sample paths) of NPeriods observations each, specified as the comma-separated pair consisting of 'NTrials' and a positive scalar integer.

Data Types: double

DeltaTimes — Positive time increments between observations

1 (default) | scalar | column vector

Positive time increments between observations, specified as the comma-separated pair consisting of 'DeltaTimes' and a scalar or an NPeriods-by-1 column vector.

DeltaTimes 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.

Data Types: double

NSteps — Number of intermediate time steps within each time increment *dt*

1 (indicating no intermediate evaluation) (default) | positive scalar integer

Number of intermediate time steps within each time increment *dt* (specified as DeltaTimes), specified as the comma-separated pair consisting of 'NSteps' and a positive scalar integer.

The simByEuler function 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.

Data Types: double

Antithetic — Flag to use antithetic sampling to generate the Gaussian random variates

false (no antithetic sampling) (default) | logical with values true or false

Flag to use antithetic sampling to generate the Gaussian random variates that drive the Brownian motion vector (Wiener processes), specified as the comma-separated pair consisting of 'Antithetic' and a scalar numeric or logical 1 (true) or 0 (false).

When you specify `true`, `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.

Note If you specify an input noise process (see `Z`), `simByEuler` ignores the value of `Antithetic`.

Data Types: `logical`

`Z` — Direct specification of the dependent random noise process for generating Brownian motion vector

generates correlated Gaussian variates based on the `Correlation` member of the `SDE` object (default) | function | three-dimensional array of dependent random variates

Direct specification of the dependent random noise process for generating the Brownian motion vector (Wiener process) that drives the simulation, specified as the comma-separated pair consisting of `'Z'` and a function or as an `(NPeriods  NSteps)`-by-`NBrowns`-by-`NTrials` three-dimensional array of dependent random variates.

Note 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` | `function`

`N` — Dependent random counting process for generating number of jumps

random numbers from Poisson distribution with parameter `JumpFreq` from a `bates` object (default) | three-dimensional array | function

Dependent random counting process for generating the number of jumps, specified as the comma-separated pair consisting of `'N'` and a function or an (`NPeriods` `NSteps`) -by-`NJumps`-by-`NTrials` three-dimensional array of dependent random variates.

Note If you specify a function, `N` must return an `NJumps`-by-`1` column vector, and you must call it with two inputs: a real-valued scalar observation time t followed by an `NVars`-by-`1` state vector X_t.

Data Types: `double` | `function`

`StorePaths` — Flag that indicates how `Paths` is stored and returned

`true` (default) | logical with values `true` or `false`

Flag that indicates how the output array `Paths` is stored and returned, specified as the comma-separated pair consisting of `'StorePaths'` and a scalar numeric or logical `1` (`true`) or `0` (`false`).

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 `Paths` as an empty matrix.

Data Types: `logical`

`MonteCarloMethod` — Monte Carlo method to simulate stochastic processes
`"standard"` (default) | string with values `"standard"`, `"quasi"`, or `"randomized-quasi"` | character vector with values `'standard'`, `'quasi'`, or `'randomized-quasi'`

Monte Carlo method to simulate stochastic processes, specified as the comma-separated pair consisting of `'MonteCarloMethod'` and a string or character vector with one of the following values:

- `"standard"` — Monte Carlo using pseudo random numbers.
- `"quasi"` — Quasi-Monte Carlo using low-discrepancy sequences.
- `"randomized-quasi"` — Randomized quasi-Monte Carlo.

Note If you specify an input noise process (see `Z` and `N`), `simByEuler` ignores the value of `MonteCarloMethod`.

Data Types: `string` | `char`

`QuasiSequence` — Low discrepancy sequence to drive the stochastic processes
`"sobol"` (default) | string with value `"sobol"` | character vector with value `'sobol'`

Low discrepancy sequence to drive the stochastic processes, specified as the comma-separated pair consisting of `'QuasiSequence'` and a string or character vector with one of the following values:

- `"sobol"` — Quasi-random low-discrepancy sequences that use a base of two to form successively finer uniform partitions of the unit interval and then reorder the coordinates in each dimension

Note If `MonteCarloMethod` option is not specified or specified as`"standard"`, `QuasiSequence` is ignored.

Data Types: `string` | `char`

`BrownianMotionMethod` — Brownian motion construction method
`"standard"` (default) | string with value `"brownian-bridge"` or `"principal-components"` | character vector with value `'brownian-bridge'` or `'principal-components'`

Brownian motion construction method, specified as the comma-separated pair consisting of `'BrownianMotionMethod'` and a string or character vector with one of the following values:

- `"standard"` — The Brownian motion path is found by taking the cumulative sum of the Gaussian variates.
- `"brownian-bridge"` — The last step of the Brownian motion path is calculated first, followed by any order between steps until all steps have been determined.
- `"principal-components"` — The Brownian motion path is calculated by minimizing the approximation error.

Note If an input noise process is specified using the Z input argument, BrownianMotionMethod is ignored.

The starting point for a Monte Carlo simulation is the construction of a Brownian motion sample path (or Wiener path). Such paths are built from a set of independent Gaussian variates, using either standard discretization, Brownian-bridge construction, or principal components construction.

Both standard discretization and Brownian-bridge construction share the same variance and therefore the same resulting convergence when used with the MonteCarloMethod using pseudo random numbers. However, the performance differs between the two when the MonteCarloMethod option "quasi" is introduced, with faster convergence seen for "brownian-bridge" construction option and the fastest convergence when using the "principal-components" construction option.

Data Types: string | char

Processes — Sequence of end-of-period processes or state vector adjustments

simByEuler makes no adjustments and performs no processing (default) | function | cell array of functions

Sequence of end-of-period processes or state vector adjustments, specified as the comma-separated pair consisting of 'Processes' and a function or cell array of functions of the form

$$X_t = P(t, X_t)$$

The simByEuler function runs processing functions at each interpolation time. The functions must accept the current interpolation time t, and the current state vector X_t and return a state vector that can 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.

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 you to define a 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).

Data Types: cell | function

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 StorePaths is set to false, simByEuler returns Paths as an empty matrix.

Times — Observation times associated with simulated paths

column vector

Observation times associated with the simulated paths, returned as an `(NPeriods + 1)`-by-`1` column vector. Each element of `Times` is associated with the corresponding row of `Paths`.

`Z` — Dependent random variates for generating Brownian motion vector
array

Dependent random variates for generating the Brownian motion vector (Wiener processes) that drive the simulation, returned as an `(NPeriods ⨉ NSteps)`-by-`NBrowns`-by-`NTrials` three-dimensional time-series array.

`N` — Dependent random variates for generating jump counting process vector
array

Dependent random variates used to generate the jump counting process vector, returned as an `(NPeriods ⨉ NSteps)`-by-`NJumps`-by-`NTrials` three-dimensional time series array.

More About

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 that has the same expected value but a 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 other pairs, 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.

Algorithms

Bates models are bivariate composite models. Each Bates model consists of two coupled univariate models.

- One model is a geometric Brownian motion (`gbm`) model with a stochastic volatility function and jumps.

  $$dX_{1t} = B(t)X_{1t}dt + \sqrt{X_{2t}}X_{1t}dW_{1t} + Y(t)X_{1t}dN_t$$

  This model usually corresponds to a price process whose volatility (variance rate) is governed by the second univariate model.
- The other model is 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 Bates price process.

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 `DeltaTimes` approaches zero.

Version History

Introduced in R2020a

R2022a: Perform Quasi-Monte Carlo simulation

Behavior changed in R2022a

Perform Quasi-Monte Carlo simulation using the name-value arguments `MonteCarloMethod` and `QuasiSequence`.

R2022b: Perform Brownian bridge and principal components construction

Behavior changed in R2022b

Perform Brownian bridge and principal components construction using the name-value argument `BrownianMotionMethod`.

References

[1] Deelstra, Griselda, and Freddy Delbaen. “Convergence of Discretized Stochastic (Interest Rate) Processes with Stochastic Drift Term.” *Applied Stochastic Models and Data Analysis*. 14, no. 1, 1998, pp. 77–84.

[2] Higham, Desmond, and Xuerong Mao. “Convergence of Monte Carlo Simulations Involving the Mean-Reverting Square Root Process.” *The Journal of Computational Finance* 8, no. 3, (2005): 35–61.

[3] Lord, Roger, Remmert Koekkoek, and Dick Van Dijk. “A Comparison of Biased Simulation Schemes for Stochastic Volatility Models.” *Quantitative Finance* 10, no. 2 (February 2010): 177–94.

See Also

`bates` | `merton`

Topics

Implementing Multidimensional Equity Market Models, Implementation 5: Using the simByEuler Method on page 14-33
“Simulating Equity Prices” on page 14-28
“Simulating Interest Rates” on page 14-48
“Stratified Sampling” on page 14-57
“Price American Basket Options Using Standard Monte Carlo and Quasi-Monte Carlo Simulation” on page 14-70
“Base SDE Models” on page 14-14
“Drift and Diffusion Models” on page 14-16
“Linear Drift Models” on page 14-19
“Parametric Models” on page 14-21
“SDEs” on page 14-2
“SDE Models” on page 14-7
“SDE Class Hierarchy” on page 14-5

“Quasi-Monte Carlo Simulation” on page 14-62
“Performance Considerations” on page 14-64

simByEuler

Simulate `Merton` jump diffusion sample paths by Euler approximation

Syntax

```
[Paths,Times,Z,N] = simByEuler(MDL,NPeriods)
[Paths,Times,Z,N] = simByEuler( ___ ,Name,Value)
```

Description

`[Paths,Times,Z,N] = simByEuler(MDL,NPeriods)` simulates `NTrials` sample paths of `NVars` correlated state variables driven by `NBrowns` Brownian motion sources of risk and `NJumps` compound Poisson processes representing the arrivals of important events over `NPeriods` consecutive observation periods. The simulation approximates the continuous-time Merton jump diffusion process by the Euler approach.

`[Paths,Times,Z,N] = simByEuler( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

You can perform quasi-Monte Carlo simulations using the name-value arguments for `MonteCarloMethod`, `QuasiSequence`, and `BrownianMotionMethod`. For more information, see “Quasi-Monte Carlo Simulation” on page 14-62.

Examples

Simulate Merton Jump Diffusion Sample Paths by Euler Approximation

Create a `merton` object.

```
AssetPrice = 80;
            Return = 0.03;
            Sigma = 0.16;
            JumpMean = 0.02;
            JumpVol = 0.08;
            JumpFreq = 2;

            mertonObj = merton(Return,Sigma,JumpFreq,JumpMean,JumpVol,...
                'startstat',AssetPrice)
```

```
mertonObj = 
   Class MERTON: Merton Jump Diffusion
   ----------------------------------------
     Dimensions: State = 1, Brownian = 1
   ----------------------------------------
      StartTime: 0
     StartState: 80
    Correlation: 1
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
```

```
            Sigma: 0.16
           Return: 0.03
         JumpFreq: 2
         JumpMean: 0.02
          JumpVol: 0.08
```

Use simByEuler to simulate NTrials sample paths of NVars correlated state variables driven by NBrowns Brownian motion sources of risk and NJumps compound Poisson processes representing the arrivals of important events over NPeriods consecutive observation periods. The function simByEuler approximates a continuous-time Merton jump diffusion process by the Euler approach.

```
NPeriods = 2;
[Paths,Times,Z,N] = simByEuler(mertonObj,NPeriods)
```

```
Paths = 3×1

   80.0000
  266.5590
  306.2600


Times = 3×1

     0
     1
     2


Z = 2×1

    1.8339
   -2.2588


N = 2×1

     1
     2
```

Paths is a 3-by-1 matrix. The only column is the path of the simulated AssetPrice. The output Z is a series of matrices used to generate the Brownian motion vector. The output N is a series of matrices used to generate jump vectors.

Quasi-Monte Carlo Simulation with simByEuler Using a Merton Model

Create a merton object.

```
AssetPrice = 80;
            Return = 0.03;
            Sigma = 0.16;
            JumpMean = 0.02;
            JumpVol = 0.08;
            JumpFreq = 2;
            mertonObj = merton(Return,Sigma,JumpFreq,JumpMean,JumpVol,...
                'startstat',AssetPrice)
```

```
mertonObj = 
   Class MERTON: Merton Jump Diffusion
   ----------------------------------------
     Dimensions: State = 1, Brownian = 1
   ----------------------------------------
      StartTime: 0
     StartState: 80
    Correlation: 1
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
          Sigma: 0.16
         Return: 0.03
       JumpFreq: 2
       JumpMean: 0.02
        JumpVol: 0.08
```

Define the quasi-Monte Carlo simulation using the optional name-value arguments for `'MonteCarloMethod'`,`'QuasiSequence'`, and `'BrownianMotionMethod'`.

```
[paths,time,z] = simByEuler(mertonObj,10,'ntrials',4096,'montecarlomethod','quasi','quasisequence
```

Input Arguments

`MDL` — Stochastic differential equation model

object

Stochastic differential equation model, specified as a `merton` object. You can create a `merton` object using `merton`.

Data Types: `object`

`NPeriods` — Number of simulation periods

positive scalar integer

Number of simulation periods, specified as a positive scalar integer. The value of `NPeriods` determines the number of rows of the simulated output series.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Paths,Times,Z,N] = simByEuler(merton,NPeriods,'DeltaTimes',dt)`

`NTrials` — Simulated trials (sample paths)

`1` (single path of correlated state variables) (default) | positive scalar integer

Simulated trials (sample paths) of NPeriods observations each, specified as the comma-separated pair consisting of 'NTrials' and a positive scalar integer.

Data Types: double

DeltaTimes — Positive time increments between observations

1 (default) | scalar | column vector

Positive time increments between observations, specified as the comma-separated pair consisting of 'DeltaTimes' and a scalar or a NPeriods-by-1 column vector.

DeltaTimes 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.

Data Types: double

NSteps — Number of intermediate time steps within each time increment

1 (indicating no intermediate evaluation) (default) | positive scalar integer

Number of intermediate time steps within each time increment *dt* (specified as DeltaTimes), specified as the comma-separated pair consisting of 'NSteps' and a positive scalar integer.

The simByEuler function 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.

Data Types: double

Antithetic — Flag to use antithetic sampling to generate the Gaussian random variates

false (no antithetic sampling) (default) | logical with values true or false

Flag to use antithetic sampling to generate the Gaussian random variates that drive the Brownian motion vector (Wiener processes), specified as the comma-separated pair consisting of 'Antithetic' and a scalar numeric or logical 1 (true) or 0 (false).

When you specify true, 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.

Note If you specify an input noise process (see Z), simByEuler ignores the value of Antithetic.

Data Types: logical

Z — Direct specification of the dependent random noise process for generating Brownian motion vector

generates correlated Gaussian variates based on the Correlation member of the SDE object (default) | function | three-dimensional array of dependent random variates

Direct specification of the dependent random noise process for generating the Brownian motion vector (Wiener process) that drives the simulation, specified as the comma-separated pair consisting of `'Z'` and a function or as an `(NPeriods ⨉ NSteps)`-by-`NBrowns`-by-`NTrials` three-dimensional array of dependent random variates.

Note 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` | `function`

`N` — Dependent random counting process for generating number of jumps
random numbers from Poisson distribution with parameter `JumpFreq` from `merton` object (default) | three-dimensional array | function

Dependent random counting process for generating the number of jumps, specified as the comma-separated pair consisting of `'N'` and a function or an (`NPeriods` ⨉ `NSteps`) -by-`NJumps`-by-`NTrials` three-dimensional array of dependent random variates.

If you specify a function, `N` must return an `NJumps`-by-`1` column vector, and you must call it with two inputs: a real-valued scalar observation time *t* followed by an `NVars`-by-`1` state vector X_t.

Data Types: `double` | `function`

`StorePaths` — Flag that indicates how `Paths` is stored and returned
`true` (default) | logical with values `true` or `false`

Flag that indicates how the output array `Paths` is stored and returned, specified as the comma-separated pair consisting of `'StorePaths'` and a scalar numeric or logical `1` (`true`) or `0` (`false`).

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 `Paths` as an empty matrix.

Data Types: `logical`

`MonteCarloMethod` — Monte Carlo method to simulate stochastic processes
`"standard"` (default) | string with values `"standard"`, `"quasi"`, or `"randomized-quasi"` | character vector with values `'standard'`, `'quasi'`, or `'randomized-quasi'`

Monte Carlo method to simulate stochastic processes, specified as the comma-separated pair consisting of `'MonteCarloMethod'` and a string or character vector with one of the following values:

- `"standard"` — Monte Carlo using pseudo random numbers.
- `"quasi"` — Quasi-Monte Carlo using low-discrepancy sequences.
- `"randomized-quasi"` — Randomized quasi-Monte Carlo.

Note If you specify an input noise process (see `Z` and `N`), `simByEuler` ignores the value of `MonteCarloMethod`.

Data Types: string | char

QuasiSequence — Low discrepancy sequence to drive the stochastic processes

"sobol" (default) | string with value "sobol" | character vector with value 'sobol'

Low discrepancy sequence to drive the stochastic processes, specified as the comma-separated pair consisting of 'QuasiSequence' and a string or character vector with one of the following values:

- "sobol" — Quasi-random low-discrepancy sequences that use a base of two to form successively finer uniform partitions of the unit interval and then reorder the coordinates in each dimension

Note If MonteCarloMethod option is not specified or specified as"standard", QuasiSequence is ignored.

Data Types: string | char

BrownianMotionMethod — Brownian motion construction method

"standard" (default) | string with value "brownian-bridge" or "principal-components" | character vector with value 'brownian-bridge' or 'principal-components'

Brownian motion construction method, specified as the comma-separated pair consisting of 'BrownianMotionMethod' and a string or character vector with one of the following values:

- "standard" — The Brownian motion path is found by taking the cumulative sum of the Gaussian variates.
- "brownian-bridge" — The last step of the Brownian motion path is calculated first, followed by any order between steps until all steps have been determined.
- "principal-components" — The Brownian motion path is calculated by minimizing the approximation error.

Note If an input noise process is specified using the Z input argument, BrownianMotionMethod is ignored.

The starting point for a Monte Carlo simulation is the construction of a Brownian motion sample path (or Wiener path). Such paths are built from a set of independent Gaussian variates, using either standard discretization, Brownian-bridge construction, or principal components construction.

Both standard discretization and Brownian-bridge construction share the same variance and therefore the same resulting convergence when used with the MonteCarloMethod using pseudo random numbers. However, the performance differs between the two when the MonteCarloMethod option "quasi" is introduced, with faster convergence seen for "brownian-bridge" construction option and the fastest convergence when using the "principal-components" construction option.

Data Types: string | char

Processes — Sequence of end-of-period processes or state vector adjustments

simByEuler makes no adjustments and performs no processing (default) | function | cell array of functions

Sequence of end-of-period processes or state vector adjustments, specified as the comma-separated pair consisting of 'Processes' and a function or cell array of functions of the form

$$X_t = P(t, X_t)$$

The `simByEuler` function runs processing functions at each interpolation time. The functions must accept the current interpolation time t, and the current state vector X_t and return a state vector that can 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.

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 you to define a `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).

Data Types: `cell` | `function`

Output Arguments

`Paths` — Simulated paths of correlated state variables
array

Simulated paths of correlated state variables, returned as an `(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 `StorePaths` is set to `false`, `simByEuler` returns `Paths` as an empty matrix.

`Times` — Observation times associated with simulated paths
column vector

Observation times associated with the simulated paths, returned as an `(NPeriods + 1)`-by-`1` column vector. Each element of `Times` is associated with the corresponding row of `Paths`.

`Z` — Dependent random variates for generating Brownian motion vector
array

Dependent random variates used to generate the Brownian motion vector (Wiener processes) that drive the simulation, returned as an `(NPeriods ⨉ NSteps)`-by-`NBrowns`-by-`NTrials` three-dimensional time-series array.

`N` — Dependent random variates used for generating jump counting process vector
array

Dependent random variates for generating the jump counting process vector, returned as an `(NPeriods ⨉ NSteps)`-by-`NJumps`-by-`NTrials` three-dimensional time-series array.

More About

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 that has the same expected value but a 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 other pairs, 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.

Algorithms

This function simulates any vector-valued SDE of the following form:

$$dX_t = B(t, X_t)X_t dt + D(t, X_t)V(t, x_t)dW_t + Y(t, X_t, N_t)X_t dN_t$$

Here:

- X_t is an `NVars`-by-`1` state vector of process variables.
- $B(t,X_t)$ is an `NVars`-by-`NVars` matrix of generalized expected instantaneous rates of return.
- $D(t,X_t)$ is an `NVars`-by-`NVars` diagonal matrix in which each element along the main diagonal is the corresponding element of the state vector.
- $V(t,X_t)$ is an `NVars`-by-`NVars` matrix of instantaneous volatility rates.
- dW_t is an `NBrowns`-by-`1` Brownian motion vector.
- $Y(t,X_t,N_t)$ is an `NVars`-by-`NJumps` matrix-valued jump size function.
- dN_t is an `NJumps`-by-`1` counting process vector.

`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.

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 `DeltaTimes` approaches zero.

Version History

Introduced in R2020a

R2022a: Perform Quasi-Monte Carlo simulation

Behavior changed in R2022a

Perform Quasi-Monte Carlo simulation using the name-value arguments `MonteCarloMethod` and `QuasiSequence`.

R2022b: Perform Brownian bridge and principal components construction

Behavior changed in R2022b

Perform Brownian bridge and principal components construction using the name-value argument `BrownianMotionMethod`.

References

[1] Deelstra, Griselda, and Freddy Delbaen. "Convergence of Discretized Stochastic (Interest Rate) Processes with Stochastic Drift Term." *Applied Stochastic Models and Data Analysis.* 14, no. 1, 1998, pp. 77–84.

[2] Higham, Desmond, and Xuerong Mao. "Convergence of Monte Carlo Simulations Involving the Mean-Reverting Square Root Process." *The Journal of Computational Finance* 8, no. 3, (2005): 35–61.

[3] Lord, Roger, Remmert Koekkoek, and Dick Van Dijk. "A Comparison of Biased Simulation Schemes for Stochastic Volatility Models." *Quantitative Finance* 10, no. 2 (February 2010): 177–94.

See Also

`bates` | `merton` | `simBySolution`

Topics

Implementing Multidimensional Equity Market Models, Implementation 5: Using the simByEuler Method on page 14-33
"Simulating Equity Prices" on page 14-28
"Simulating Interest Rates" on page 14-48
"Stratified Sampling" on page 14-57
"Price American Basket Options Using Standard Monte Carlo and Quasi-Monte Carlo Simulation" on page 14-70
"Base SDE Models" on page 14-14
"Drift and Diffusion Models" on page 14-16
"Linear Drift Models" on page 14-19
"Parametric Models" on page 14-21
"SDEs" on page 14-2
"SDE Models" on page 14-7
"SDE Class Hierarchy" on page 14-5
"Quasi-Monte Carlo Simulation" on page 14-62
"Performance Considerations" on page 14-64

simBySolution

Simulate approximate solution of diagonal-drift Merton jump diffusion process

Syntax

```
[Paths,Times,Z,N] = simBySolution(MDL,NPeriods)
[Paths,Times,Z,N] = simBySolution( ___ ,Name,Value)
```

Description

[Paths,Times,Z,N] = simBySolution(MDL,NPeriods) simulates NNTrials sample paths of NVars correlated state variables driven by NBrowns Brownian motion sources of risk and NJumps compound Poisson processes representing the arrivals of important events over NPeriods consecutive observation periods. The simulation approximates continuous-time Merton jump diffusion process by an approximation of the closed-form solution.

[Paths,Times,Z,N] = simBySolution(___ ,Name,Value) specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

You can perform quasi-Monte Carlo simulations using the name-value arguments for MonteCarloMethod, QuasiSequence, and BrownianMotionMethod. For more information, see "Quasi-Monte Carlo Simulation" on page 14-62.

Examples

Use simBySolution with merton Object

Simulate the approximate solution of diagonal-drift Merton process.

Create a merton object.

```
AssetPrice = 80;
            Return = 0.03;
            Sigma = 0.16;
            JumpMean = 0.02;
            JumpVol = 0.08;
            JumpFreq = 2;

            mertonObj = merton(Return,Sigma,JumpFreq,JumpMean,JumpVol,...
                'startstat',AssetPrice)
```

```
mertonObj = 
   Class MERTON: Merton Jump Diffusion
   ----------------------------------------
     Dimensions: State = 1, Brownian = 1
   ----------------------------------------
      StartTime: 0
     StartState: 80
    Correlation: 1
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
          Sigma: 0.16
         Return: 0.03
       JumpFreq: 2
       JumpMean: 0.02
        JumpVol: 0.08
```

Use simBySolution to simulate NTrials sample paths of NVARS correlated state variables driven by NBrowns Brownian motion sources of risk and NJumps compound Poisson processes representing the arrivals of important events over NPeriods consecutive observation periods. The function approximates continuous-time Merton jump diffusion process by an approximation of the closed-form solution.

```
nPeriods = 100;
[Paths,Times,Z,N] = simBySolution(mertonObj, nPeriods,'nTrials', 3)
```

```
Paths = 
Paths(:,:,1) =

   1.0e+03 *

    0.0800
    0.0600
    0.0504
    0.0799
    0.1333
    0.1461
    0.2302
    0.2505
    0.3881
    0.4933
    0.4547
    0.4433
    0.5294
    0.6443
    0.7665
    0.6489
    0.7220
    0.7110
    0.5815
    0.5026
    0.6523
    0.7005
    0.7053
    0.4902
    0.5401
    0.4730
    0.4242
    0.5334
    0.5821
    0.6498
    0.5982
    0.5504
    0.5290
    0.5371
    0.4789
    0.4914
    0.5019
    0.3557
    0.2950
    0.3697
    0.2906
    0.2988
    0.3081
    0.3469
    0.3146
    0.3171
    0.3588
    0.3250
    0.3035
    0.2386
    0.2533
    0.2420
    0.2315
    0.2396
    0.2143
    0.2668
    0.2115
    0.1671
    0.1784
    0.1542
    0.2046
    0.1930
    0.2011
    0.2542
    0.3010
    0.3247
    0.3900
    0.4107
    0.3949
    0.4610
    0.5725
    0.5605
    0.4541
    0.5796
    0.8199
    0.5732
    0.5856
    0.7895
    0.6883
    0.6848
    0.9059
    1.0089
    0.8429
    0.9955
    0.9683
    0.8769
    0.7120
    0.7906
    0.7630
    1.2460
    1.1703
    1.2012
    1.1109
    1.1893
    1.4346
    1.4040
    1.2365
    1.0834
    1.3315
    0.8100
    0.5558


Paths(:,:,2) =

   80.0000
   81.2944
   71.3663
  108.8305
  111.4851
  105.4563
  160.2721
  125.3288
  158.3238
  138.8899
  157.9613
  125.6819
  149.8234
  126.0374
  182.5153
  195.0861
  273.1622
  306.2727
  301.3401
  312.2173
  298.2344
  327.6944
  288.9799
  394.8951
  551.4020
  418.2258
  404.1687
  469.3555
  606.4289
  615.7066
  526.6862
  625.9683
  474.4597
  316.5110
  407.9626
  341.6552
  475.0593
  478.4058
  545.3414
  365.3404
  513.2186
  370.5371
  444.0345
  314.6991
  257.4782
  253.0259
  237.6185
  206.6325
  334.5253
  300.2284
  328.9936
  307.4059
  248.7966
  234.6355
  183.9132
  159.6084
  169.1145
  123.3256
  148.1922
  159.7083
  104.0447
   96.3935
   92.4897
   93.0576
  116.3163
  135.6249
  120.6611
  100.0253
  109.7998
   85.8078
   81.5769
   73.7983
   65.9000
   62.5120
   62.9952
   57.6044
   54.2716
   44.5617
   42.2402
   21.9133
   18.0586
   20.5171
   22.5532
   24.1654
   26.8830
   22.7864
   34.5131
   27.8362
   27.7258
   21.7367
   20.8781
   19.7174
   14.9880
   14.8903
   19.3632
   23.4230
   27.7062
   17.8347
   16.8652
   15.5675
   15.5256


Paths(:,:,3) =
   80.0000
   79.6263
   93.2979
   63.1451
   60.2213
   54.2113
   78.6114
   96.6261
  123.5584
  126.5875
  102.9870
   83.2387
   77.8567
   79.3565
   71.3876
   80.5413
   90.8709
   77.5246
  107.4194
  114.4328
  118.3999
  148.0710
  108.6207
  110.0402
  124.1150
  104.5409
   94.7576
   98.9002
  108.0691
  130.7592
  129.9744
  119.9150
   86.0303
   96.9892
   86.8928
  106.8895
  119.3219
  197.7045
  208.1930
  197.1636
  244.4438
  166.4752
  125.3896
  128.9036
  170.9818
  140.2719
  125.8948
   87.0324
   66.7637
   48.4280
   50.5766
   49.7841
   67.5690
   62.8776
   85.3896
   67.9608
   72.9804
   59.0174
   50.1132
   45.2220
   59.5469
   58.4673
   98.4790
   90.0250
   80.3092
   86.9245
   88.1303
   95.4237
  104.4456
   99.1969
  168.3980
  146.8791
  150.0052
  129.7521
  127.1402
  113.3413
  145.2281
  153.1315
  125.7882
  111.9988
  112.7732
  118.9120
  150.9166
  120.0673
  128.2727
  185.9171
  204.3474
  194.5443
  163.2626
  183.9897
  233.4125
  318.9068
  356.0077
  380.4513
  446.9518
  484.9218
  377.4244
  470.3577
  454.5734
  297.0580
  339.0796

Times = 101×1

     0
     1
     2
     3
     4
     5
     6
     7
     8
     9
```

```
      ⋮

   Z =
   Z(:,:,1) =

      -2.2588
      -1.3077
       3.5784
       3.0349
       0.7147
       1.4897
       0.6715
       1.6302
       0.7269
      -0.7873
      -1.0689
       1.4384
       1.3703
      -0.2414
      -0.8649
       0.6277
      -0.8637
      -1.1135
      -0.7697
       1.1174
       0.5525
       0.0859
      -1.0616
       0.7481
      -0.7648
       0.4882
       1.4193
       1.5877
       0.8351
      -1.1658
       0.7223
       0.1873
      -0.4390
      -0.8880
       0.3035
       0.7394
      -2.1384
      -1.0722
       1.4367
      -1.2078
       1.3790
      -0.2725
       0.7015
      -0.8236
       0.2820
       1.1275
       0.0229
      -0.2857
      -1.1564
       0.9642
      -0.0348
      -0.1332
      -0.2248
      -0.8479
       1.6555
      -0.8655
      -1.3320
       0.3335
      -0.1303
       0.8620
      -0.8487
       1.0391
       0.6601
      -0.2176
       0.0513
       0.4669
       0.1832
       0.3071
       0.2614
      -0.1461
      -0.8757
      -1.1742
       1.5301
       1.6035
      -1.5062
       0.2761
       0.3919
      -0.7411
       0.0125
       1.2424
       0.3503
      -1.5651
       0.0983
      -0.0308
      -0.3728
      -2.2584
       1.0001
      -0.2781
       0.4716
       0.6524
       1.0061
      -0.9444
       0.0000
       0.5946
       0.9298
      -0.6516
      -0.0245
       0.8617
      -2.4863
      -2.3193


   Z(:,:,2) =

       0.8622
      -0.4336
       2.7694
       0.7254
      -0.2050
       1.4090
```

```
  -1.2075
   0.4889
  -0.3034
   0.8884
  -0.8095
   0.3252
  -1.7115
   0.3192
  -0.0301
   1.0933
   0.0774
  -0.0068
   0.3714
  -1.0891
   1.1006
  -1.4916
   2.3505
  -0.1924
  -1.4023
  -0.1774
   0.2916
  -0.8045
  -0.2437
  -1.1480
   2.5855
  -0.0825
  -1.7947
   0.1001
  -0.6003
   1.7119
  -0.8396
   0.9610
  -1.9609
   2.9080
  -1.0582
   1.0984
  -2.0518
  -1.5771
   0.0335
   0.3502
  -0.2620
  -0.8314
  -0.5336
   0.5201
  -0.7982
  -0.7145
  -0.5890
  -1.1201
   0.3075
  -0.1765
  -2.3299
   0.3914
   0.1837
  -1.3617
  -0.3349
  -1.1176
  -0.0679
  -0.3031
```

```
    0.8261
   -0.2097
   -1.0298
    0.1352
   -0.9415
   -0.5320
   -0.4838
   -0.1922
   -0.2490
    1.2347
   -0.4446
   -0.2612
   -1.2507
   -0.5078
   -3.0292
   -1.0667
   -0.0290
   -0.0845
    0.0414
    0.2323
   -0.2365
    2.2294
   -1.6642
    0.4227
   -1.2128
    0.3271
   -0.6509
   -1.3218
   -0.0549
    0.3502
    0.2398
    1.1921
   -1.9488
    0.0012
    0.5812
    0.0799


Z(:,:,3) =

    0.3188
    0.3426
   -1.3499
   -0.0631
   -0.1241
    1.4172
    0.7172
    1.0347
    0.2939
   -1.1471
   -2.9443
   -0.7549
   -0.1022
    0.3129
   -0.1649
    1.1093
   -1.2141
    1.5326
   -0.2256
    0.0326
    1.5442
   -0.7423
   -0.6156
    0.8886
   -1.4224
   -0.1961
    0.1978
    0.6966
    0.2157
    0.1049
   -0.6669
   -1.9330
    0.8404
   -0.5445
    0.4900
   -0.1941
    1.3546
    0.1240
   -0.1977
    0.8252
   -0.4686
   -0.2779
   -0.3538
    0.5080
   -1.3337
   -0.2991
   -1.7502
   -0.9792
   -2.0026
   -0.0200
    1.0187
    1.3514
   -0.2938
    2.5260
   -1.2571
    0.7914
   -1.4491
    0.4517
   -0.4762
    0.4550
    0.5528
    1.2607
   -0.1952
    0.0230
    1.5270
    0.6252
    0.9492
    0.5152
   -0.1623
    1.6821
   -0.7120
   -0.2741
   -1.0642
   -0.2296
   -0.1559
    0.4434
   -0.9480
   -0.3206
   -0.4570
    0.9337
    0.1825
    1.6039
   -0.7342
    0.4264
    2.0237
    0.3376
   -0.5900
   -1.6702
    0.0662
    1.0826
    0.2571
    0.9248
    0.9111
    1.2503
   -0.6904
   -1.6118
    1.0205
   -0.0708
   -2.1924
   -0.9485


N =
N(:,:,1) =

     3
     1
     2
     1
     0
     2
     0
     1
     3
     4
     2
     1
     0
     1
     1
     1
     1
     0
     0
     3
     2
     2
     1
     0
     1
     1
     3
     3
     4
     2
     4
     1
     1
     2
     0
     2
     2
     3
     2
     1
     3
     2
     2
     1
     1
     1
     3
     0
     2
     2
     1
     0
     1
     1
     1
     1
     0
     2
     2
     1
     1
     5
     7
     3
     2
     2
     1
     3
     3
     5
     3
     0
     1
     6
     2
     0
     5
     2
     2
     1
     2
     1
     3
     0
     2
     4
     2
     2
     4
     2
     3
     1
     2
     5
     1
     0
     3
     3
     1
     1


N(:,:,2) =

     4
     2
     2
     2
     0
     4
     1
     2
     3
     1
     2
     1
     4
     2
     6
     2
     2
     2
     2
     1
     4
     3
     1
     3
     3
     1
     3
     6
     1
     4
     2
     2
     1
     2
     1
     1
     5
     0
     2
     2
     3
     2
     2
     1
     0
     1
     5
     4
     0
     1
     1
     2
     1
     2
     3
     2
     2
     1
     2
     2
     0
     3
     1
     6
     3
     3
     0
     2
     1
     2
     0
     6
     1
     3
     1
     2
     2
     2
     1
     0
     2
     2
     2
     2
     1
     1
     3
     1
     2
     2
     1
     4
     1
     3
     3
     0
     1
     1
     1
     2


N(:,:,3) =

     1
     3
     2
     2
     1
     4
     2
     3
     0
     0
     4
     3
     2
     3
     1
     1
     1
     1
     3
     4
     1
     2
     3
     1
     1
     1
     1
     0
     3
     0
     1
     0
     4
     0
     2
     4
     3
     1
     0
     1
     5
     3
     3
     2
     1
     2
     3
     1
     5
     4
     1
     1
     2
     2
     1
     1
     1
     2
     1
     5
     1
     2
     1
     3
     2
     2
     1
     3
     1
     6
     0
     1
     4
     1
     1
     3
     5
     3
     1
     2
     2
     1
     2
     1
     1
     1
     1
     1
     2
     3
     6
     2
     1
     3
     2
     1
     1
     0
     1
     3
```

Quasi-Monte Carlo Simulation Using Merton Model

This example shows how to use `simBySolution` with a Merton model to perform a quasi-Monte Carlo simulation. Quasi-Monte Carlo simulation is a Monte Carlo simulation that uses quasi-random sequences instead pseudo random numbers.

Create a `merton` object.

```
AssetPrice = 80;
Return = 0.03;
Sigma = 0.16;
JumpMean = 0.02;
JumpVol = 0.08;
JumpFreq = 2;

Merton = merton(Return,Sigma,JumpFreq,JumpMean,JumpVol,'startstat',AssetPrice)
```

```
Merton = 
   Class MERTON: Merton Jump Diffusion
   ----------------------------------------
     Dimensions: State = 1, Brownian = 1
   ----------------------------------------
      StartTime: 0
     StartState: 80
    Correlation: 1
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
          Sigma: 0.16
         Return: 0.03
       JumpFreq: 2
       JumpMean: 0.02
        JumpVol: 0.08
```

Perform a quasi-Monte Carlo simulation by using `simBySolution` with the optional name-value arguments for `'MonteCarloMethod'`,`'QuasiSequence'`, and `'BrownianMotionMethod'`.

```
[paths,time,z,n] = simBySolution(Merton, 10,'ntrials',4096,'montecarlomethod','quasi','QuasiSeque
```

Input Arguments

`MDL` — Merton model

`merton` object

Merton model, specified as a `merton` object. You can create a `merton` object using `merton`.

Data Types: `object`

`NPeriods` — Number of simulation periods

positive integer

Number of simulation periods, specified as a positive scalar integer. The value of `NPeriods` determines the number of rows of the simulated output series.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Paths,Times,Z,N] = simBySolution(merton,NPeriods,'DeltaTimes',dt,'NNTrials',10)`

`NNTrials` — Simulated NTrials (sample paths)

`1` (single path of correlated state variables) (default) | positive integer

Simulated NTrials (sample paths) of `NPeriods` observations each, specified as the comma-separated pair consisting of `'NNTrials'` and a positive scalar integer.

Data Types: `double`

`DeltaTimes` — Positive time increments between observations

`1` (default) | scalar | column vector

Positive time increments between observations, specified as the comma-separated pair consisting of `'DeltaTimes'` and a scalar or an `NPeriods`-by-`1` column vector.

`DeltaTimes` 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.

Data Types: `double`

`NSteps` — Number of intermediate time steps within each time increment

`1` (indicating no intermediate evaluation) (default) | positive integer

Number of intermediate time steps within each time increment *dt* (specified as `DeltaTimes`), specified as the comma-separated pair consisting of `'NSteps'` and a positive scalar integer.

The `simBySolution` function 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.

Data Types: `double`

`Antithetic` — Flag to use antithetic sampling to generate the Gaussian random variates

`false` (no antithetic sampling) (default) | logical with values `true` or `false`

Flag to use antithetic sampling to generate the Gaussian random variates that drive the Brownian motion vector (Wiener processes), specified as the comma-separated pair consisting of `'Antithetic'` and a scalar numeric or logical `1` (`true`) or `0` (`false`).

When you specify `true`, `simBySolution` performs sampling such that all primary and antithetic paths are simulated and stored in successive matching pairs:

- Odd NTrials `(1,3,5,...)` correspond to the primary Gaussian paths.
- Even NTrials `(2,4,6,...)` are the matching antithetic paths of each pair derived by negating the Gaussian draws of the corresponding primary (odd) trial.

Note If you specify an input noise process (see `Z`), `simBySolution` ignores the value of `Antithetic`.

Data Types: `logical`

MonteCarloMethod — Monte Carlo method to simulate stochastic processes
"standard" (default) | string with values "standard", "quasi", or "randomized-quasi" | character vector with values 'standard', 'quasi', or 'randomized-quasi'

Monte Carlo method to simulate stochastic processes, specified as the comma-separated pair consisting of 'MonteCarloMethod' and a string or character vector with one of the following values:

- "standard" — Monte Carlo using pseudo random numbers.
- "quasi" — Quasi-Monte Carlo using low-discrepancy sequences.
- "randomized-quasi" — Randomized quasi-Monte Carlo.

Note If you specify an input noise process (see Z and N), simBySolution ignores the value of MonteCarloMethod.

Data Types: string | char

QuasiSequence — Low discrepancy sequence to drive the stochastic processes
"sobol" (default) | string with value "sobol" | character vector with value 'sobol'

Low discrepancy sequence to drive the stochastic processes, specified as the comma-separated pair consisting of 'QuasiSequence' and a string or character vector with one of the following values:

- "sobol" — Quasi-random low-discrepancy sequences that use a base of two to form successively finer uniform partitions of the unit interval and then reorder the coordinates in each dimension

Note

- If MonteCarloMethod option is not specified or specified as"standard", QuasiSequence is ignored.
- If you specify an input noise process (see Z), simBySolution ignores the value of QuasiSequence.

Data Types: string | char

BrownianMotionMethod — Brownian motion construction method
"standard" (default) | string with value "brownian-bridge" or "principal-components" | character vector with value 'brownian-bridge' or 'principal-components'

Brownian motion construction method, specified as the comma-separated pair consisting of 'BrownianMotionMethod' and a string or character vector with one of the following values:

- "standard" — The Brownian motion path is found by taking the cumulative sum of the Gaussian variates.
- "brownian-bridge" — The last step of the Brownian motion path is calculated first, followed by any order between steps until all steps have been determined.
- "principal-components" — The Brownian motion path is calculated by minimizing the approximation error.

Note If an input noise process is specified using the `Z` input argument, `BrownianMotionMethod` is ignored.

The starting point for a Monte Carlo simulation is the construction of a Brownian motion sample path (or Wiener path). Such paths are built from a set of independent Gaussian variates, using either standard discretization, Brownian-bridge construction, or principal components construction.

Both standard discretization and Brownian-bridge construction share the same variance and therefore the same resulting convergence when used with the `MonteCarloMethod` using pseudo random numbers. However, the performance differs between the two when the `MonteCarloMethod` option `"quasi"` is introduced, with faster convergence seen for `"brownian-bridge"` construction option and the fastest convergence when using the `"principal-components"` construction option.

Data Types: `string` | `char`

Z — Direct specification of the dependent random noise process for generating Brownian motion vector

generates correlated Gaussian variates based on the `Correlation` member of the `SDE` object (default) | function | three-dimensional array of dependent random variates

Direct specification of the dependent random noise process for generating the Brownian motion vector (Wiener process) that drives the simulation, specified as the comma-separated pair consisting of `'Z'` and a function or an `(NPeriods * NSteps)`-by-`NBrowns`-by-`NNTrials` three-dimensional array of dependent random variates.

The input argument `Z` allows you to directly specify the noise generation process. This process takes precedence over the `Correlation` parameter of the input `merton` object and the value of the `Antithetic` input flag.

Specifically, when `Z` is specified, `Correlation` is not explicitly used to generate the Gaussian variates that drive the Brownian motion. However, `Correlation` is still used in the expression that appears in the exponential term of the log[X_t] Euler scheme. Thus, you must specify `Z` as a correlated Gaussian noise process whose correlation structure is consistently captured by `Correlation`.

Note 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` | `function`

N — Dependent random counting process for generating number of jumps

random numbers from Poisson distribution with `merton` object parameter `JumpFreq` (default) | three-dimensional array | function

Dependent random counting process for generating the number of jumps, specified as the comma-separated pair consisting of `'N'` and a function or an (`NPeriods` □ `NSteps`) -by-`NJumps`-by-`NNTrials` three-dimensional array of dependent random variates. If you specify a function, `N` must return an `NJumps`-by-`1` column vector, and you must call it with two inputs: a real-valued scalar observation time t followed by an `NVars`-by-`1` state vector X_t.

Data Types: double | function

StorePaths — Flag that indicates how Paths is stored and returned
true (default) | logical with values true or false

Flag that indicates how the output array Paths is stored and returned, specified as the comma-separated pair consisting of 'StorePaths' and a scalar numeric or logical 1 (true) or 0 (false).

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 Paths as an empty matrix.

Data Types: logical

Processes — Sequence of end-of-period processes or state vector adjustments of the form
simBySolution makes no adjustments and performs no processing (default) | function | cell array of functions

Sequence of end-of-period processes or state vector adjustments, specified as the comma-separated pair consisting of 'Processes' and a function or cell array of functions of the form

$$X_t = P(t, X_t)$$

simBySolution applies processing functions at the end of each observation period. These functions must accept the current observation time t and the current state vector X_t, and return a state vector that can be an adjustment to the input state.

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).

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: cell | function

Output Arguments

Paths — Simulated paths of correlated state variables
array

Simulated paths of correlated state variables, returned as an (NPeriods + 1)-by-NVars-by-NNTrials 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 StorePaths is set to false, simBySolution returns Paths as an empty matrix.

Times — Observation times associated with simulated paths
column vector

Observation times associated with the simulated paths, returned as an (NPeriods + 1)-by-1 column vector. Each element of Times is associated with the corresponding row of Paths.

Z — Dependent random variates for generating the Brownian motion vector

array

Dependent random variates for generating the Brownian motion vector (Wiener processes) that drive the simulation, returned as a `(NPeriods * NSteps)`-by-`NBrowns`-by-`NNTrials` three-dimensional time-series array.

N — Dependent random variates for generating the jump counting process vector

array

Dependent random variates for generating the jump counting process vector, returned as an `(NPeriods □ NSteps)`-by-`NJumps`-by-`NNTrials` three-dimensional time-series array.

More About

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 that has the same expected value but a 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 other pairs, 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 NTrials.

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.

Algorithms

The `simBySolution` function simulates the state vector X_t by an approximation of the closed-form solution of diagonal drift Merton jump diffusion models. Specifically, it applies a Euler approach to the transformed `log`[X_t] process (using Ito's formula). In general, this is not the exact solution to the Merton jump diffusion model because the probability distributions of the simulated and true state vectors are identical only for piecewise constant parameters.

This function simulates any vector-valued `merton` process of the form

$$dX_t = B(t, X_t)X_tdt + D(t, X_t)V(t, x_t)dW_t + Y(t, X_t, N_t)X_tdN_t$$

Here:

- X_t is an `NVars`-by-`1` state vector of process variables.
- $B(t,X_t)$ is an `NVars`-by-`NVars` matrix of generalized expected instantaneous rates of return.
- $D(t,X_t)$ is an `NVars`-by-`NVars` diagonal matrix in which each element along the main diagonal is the corresponding element of the state vector.
- $V(t,X_t)$ is an `NVars`-by-`NVars` matrix of instantaneous volatility rates.

- dW_t is an `NBrowns`-by-`1` Brownian motion vector.
- $Y(t,X_t,N_t)$ is an `NVars`-by-`NJumps` matrix-valued jump size function.
- dN_t is an `NJumps`-by-`1` counting process vector.

Version History

Introduced in R2020a

R2022a: Perform Quasi-Monte Carlo simulation

Behavior changed in R2022a

Perform Quasi-Monte Carlo simulation using the name-value arguments `MonteCarloMethod` and `QuasiSequence`.

R2022b: Perform Brownian bridge and principal components construction

Behavior changed in R2022b

Perform Brownian bridge and principal components construction using the name-value argument `BrownianMotionMethod`.

References

[1] Aït-Sahalia, Yacine. “Testing Continuous-Time Models of the Spot Interest Rate.” *Review of Financial Studies* 9, no. 2 (Apr. 1996): 385–426.

[2] Aït-Sahalia, Yacine. “Transition Densities for Interest Rate and Other Nonlinear Diffusions.” *The Journal of Finance* 54, no. 4 (Aug. 1999): 1361–95.

[3] Glasserman, Paul. *Monte Carlo Methods in Financial Engineering*. New York: Springer-Verlag, 2004.

[4] Hull, John C. *Options, Futures and Other Derivatives*. 7th ed, Prentice Hall, 2009.

[5] Johnson, Norman Lloyd, Samuel Kotz, and Narayanaswamy Balakrishnan. *Continuous Univariate Distributions*. 2nd ed. Wiley Series in Probability and Mathematical Statistics. New York: Wiley, 1995.

[6] Shreve, Steven E. *Stochastic Calculus for Finance*. New York: Springer-Verlag, 2004.

See Also

`simByEuler` | `merton`

Topics

“Simulating Equity Prices” on page 14-28
“Simulating Interest Rates” on page 14-48
“Stratified Sampling” on page 14-57
“Price American Basket Options Using Standard Monte Carlo and Quasi-Monte Carlo Simulation” on page 14-70
“Base SDE Models” on page 14-14
“Drift and Diffusion Models” on page 14-16
“Linear Drift Models” on page 14-19
“Parametric Models” on page 14-21

“SDEs” on page 14-2
“SDE Models” on page 14-7
“SDE Class Hierarchy” on page 14-5
“Quasi-Monte Carlo Simulation” on page 14-62
“Performance Considerations” on page 14-64

simByQuadExp

Simulate Bates, Heston, and CIR sample paths by quadratic-exponential discretization scheme

Syntax

```
[Paths,Times,Z] = simByQuadExp(MDL,NPeriods)
[Paths,Times,Z] = simByQuadExp( ___ ,Name,Value)

[Paths,Times,Z,N] = simByQuadExp(MDL,NPeriods)
[Paths,Times,Z,N] = simByQuadExp( ___ ,Name,Value)
```

Description

[Paths,Times,Z] = simByQuadExp(MDL,NPeriods) simulates NTrials sample paths of a Heston model driven by two Brownian motion sources of risk, or a CIR model driven by one Brownian motion source of risk. Both Heston and Bates models approximate continuous-time stochastic processes by a quadratic-exponential discretization scheme. The simByQuadExp simulation derives directly from the stochastic differential equation of motion; the discrete-time process approaches the true continuous-time process only in the limit as DeltaTimes approaches zero.

[Paths,Times,Z] = simByQuadExp(___ ,Name,Value) specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

[Paths,Times,Z,N] = simByQuadExp(MDL,NPeriods) simulates NTrials sample paths of a Bates model driven by two Brownian motion sources of risk, approximating continuous-time stochastic processes by a quadratic-exponential discretization scheme. The simByQuadExp simulation derives directly from the stochastic differential equation of motion; the discrete-time process approaches the true continuous-time process only in the limit as DeltaTimes approaches zero.

[Paths,Times,Z,N] = simByQuadExp(___ ,Name,Value) specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

You can perform quasi-Monte Carlo simulations using the name-value arguments for MonteCarloMethod, QuasiSequence, and BrownianMotionMethod. For more information, see “Quasi-Monte Carlo Simulation” on page 14-62.

Examples

Simulate Bates Sample Paths by Quadratic-Exponential Discretization Scheme

Create a bates object.

```
AssetPrice = 80;
            Return = 0.03;
            JumpMean = 0.02;
            JumpVol = 0.08;
            JumpFreq = 0.1;
            V0 = 0.04;
            Level = 0.05;
            Speed = 1.0;
            Volatility = 0.2;
            Rho = -0.7;
            StartState = [AssetPrice;V0];
            Correlation = [1 Rho;Rho 1];

batesObj = bates(Return, Speed, Level, Volatility,...
                JumpFreq, JumpMean, JumpVol,'startstate',StartState,...
                'correlation',Correlation)
```

```
batesObj = 
   Class BATES: Bates Bivariate Stochastic Volatility
   --------------------------------------------------
     Dimensions: State = 2, Brownian = 2
   --------------------------------------------------
      StartTime: 0
     StartState: 2x1 double array
    Correlation: 2x2 double array
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
         Return: 0.03
          Speed: 1
          Level: 0.05
     Volatility: 0.2
       JumpFreq: 0.1
       JumpMean: 0.02
        JumpVol: 0.08
```

Use simByQuadExp to simulate NTrials sample paths directly from the stochastic differential equation of motion; the discrete-time process approaches the true continuous-time process only in the limit as DeltaTimes approaches zero.

```
NPeriods = 2;
[Paths,Times,Z,N] = simByQuadExp(batesObj,NPeriods)
```

```
Paths = 3×2

   80.0000    0.0400
   64.3377    0.1063
   31.5703    0.1009


Times = 3×1

     0
     1
     2


Z = 2×2

    0.5377    1.8339
   -2.2588    0.8622
```

```
N = 2×1

     0
     0
```

The output Paths is returned as a (NPeriods + 1)-by-NVars-by-NTrials three-dimensional time-series array.

Quasi-Monte Carlo Simulation Using Bates Model

This example shows how to use simByQuadExp with a Bates model to perform a quasi-Monte Carlo simulation. Quasi-Monte Carlo simulation is a Monte Carlo simulation that uses quasi-random sequences instead pseudo random numbers.

Define the parameters for the bates object.

```
AssetPrice = 80;
Return = 0.03;
JumpMean = 0.02;
JumpVol = 0.08;
JumpFreq = 0.1;

V0 = 0.04;
Level = 0.05;
Speed = 1.0;
Volatility = 0.2;
Rho = -0.7;
StartState = [AssetPrice;V0];
Correlation = [1 Rho;Rho 1];
```

Create a bates object.

```
Bates = bates(Return, Speed, Level, Volatility, ...
                JumpFreq, JumpMean, JumpVol,'startstate',StartState, ...
                'correlation',Correlation)
```

```
Bates = 
   Class BATES: Bates Bivariate Stochastic Volatility
   --------------------------------------------------
     Dimensions: State = 2, Brownian = 2
   --------------------------------------------------
      StartTime: 0
     StartState: 2x1 double array
    Correlation: 2x2 double array
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
         Return: 0.03
          Speed: 1
          Level: 0.05
     Volatility: 0.2
       JumpFreq: 0.1
       JumpMean: 0.02
        JumpVol: 0.08
```

Perform a quasi-Monte Carlo simulation by using `simByQuadExp` with the optional name-value arguments for `'MonteCarloMethod'`,`'QuasiSequence'`, and `'BrownianMotionMethod'`.

```
[paths,time,z] = simByQuadExp(Bates,10,'ntrials',4096,'montecarlomethod','quasi','quasisequence'
```

Input Arguments

`MDL` — Stochastic differential equation model

bates object | heston object | cir object

Stochastic differential equation model, specified as a `bates`, `heston`, or `cir` object. You can create these objects using `bates`, `heston`, or `cir`.

Data Types: `object`

`NPeriods` — Number of simulation periods

positive scalar integer

Number of simulation periods, specified as a positive scalar integer. The value of `NPeriods` determines the number of rows of the simulated output series.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Paths,Times,Z,N] = simByQuadExp(bates_obj,NPeriods,'DeltaTime',dt)`

`NTrials` — Simulated trials (sample paths) of `NPeriods` observations each

`1` (single path of correlated state variables) (default) | positive scalar integer

Simulated trials (sample paths) of `NPeriods` observations each, specified as the comma-separated pair consisting of `'NTrials'` and a positive scalar integer.

Data Types: `double`

`DeltaTimes` — Positive time increments between observations

`1` (default) | scalar | column vector

Positive time increments between observations, specified as the comma-separated pair consisting of `'DeltaTimes'` and a scalar or a `NPeriods`-by-`1` column vector.

`DeltaTimes` 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.

Data Types: `double`

`NSteps` — Number of intermediate time steps within each time increment *dt* (specified as `DeltaTimes`)

`1` (indicating no intermediate evaluation) (default) | positive scalar integer

Number of intermediate time steps within each time increment *dt* (specified as `DeltaTimes`), specified as the comma-separated pair consisting of `'NSteps'` and a positive scalar integer.

The `simByQuadExp` function 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 `simByQuadExp` 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.

Data Types: `double`

`MonteCarloMethod` — Monte Carlo method to simulate stochastic processes
`"standard"` (default) | string with values `"standard"`, `"quasi"`, or `"randomized-quasi"` | character vector with values `'standard'`, `'quasi'`, or `'randomized-quasi'`

Monte Carlo method to simulate stochastic processes, specified as the comma-separated pair consisting of `'MonteCarloMethod'` and a string or character vector with one of the following values:

- `"standard"` — Monte Carlo using pseudo random numbers.
- `"quasi"` — Quasi-Monte Carlo using low-discrepancy sequences.
- `"randomized-quasi"` — Randomized quasi-Monte Carlo.

Note If you specify an input noise process (see `Z` and `N`), `simByQuadExp` ignores the value of `MonteCarloMethod`.

Data Types: `string` | `char`

`QuasiSequence` — Low discrepancy sequence to drive the stochastic processes
`"sobol"` (default) | string with value `"sobol"` | character vector with value `'sobol'`

Low discrepancy sequence to drive the stochastic processes, specified as the comma-separated pair consisting of `'QuasiSequence'` and a string or character vector with one of the following values:

- `"sobol"` — Quasi-random low-discrepancy sequences that use a base of two to form successively finer uniform partitions of the unit interval and then reorder the coordinates in each dimension

Note If `MonteCarloMethod` option is not specified or specified as`"standard"`, `QuasiSequence` is ignored.

Data Types: `string` | `char`

`BrownianMotionMethod` — Brownian motion construction method
`"standard"` (default) | string with value `"brownian-bridge"` or `"principal-components"` | character vector with value `'brownian-bridge'` or `'principal-components'`

Brownian motion construction method, specified as the comma-separated pair consisting of `'BrownianMotionMethod'` and a string or character vector with one of the following values:

- `"standard"` — The Brownian motion path is found by taking the cumulative sum of the Gaussian variates.

- `"brownian-bridge"` — The last step of the Brownian motion path is calculated first, followed by any order between steps until all steps have been determined.
- `"principal-components"` — The Brownian motion path is calculated by minimizing the approximation error.

Note If an input noise process is specified using the `Z` input argument, `BrownianMotionMethod` is ignored.

The starting point for a Monte Carlo simulation is the construction of a Brownian motion sample path (or Wiener path). Such paths are built from a set of independent Gaussian variates, using either standard discretization, Brownian-bridge construction, or principal components construction.

Both standard discretization and Brownian-bridge construction share the same variance and therefore the same resulting convergence when used with the `MonteCarloMethod` using pseudo random numbers. However, the performance differs between the two when the `MonteCarloMethod` option `"quasi"` is introduced, with faster convergence seen for `"brownian-bridge"` construction option and the fastest convergence when using the `"principal-components"` construction option.

Data Types: `string` | `char`

`Antithetic` — Flag to use antithetic sampling to generate the Gaussian random variates

`false` (no antithetic sampling) (default) | logical with values `true` or `false`

Flag to use antithetic sampling to generate the Gaussian random variates that drive the Brownian motion vector (Wiener processes), specified as the comma-separated pair consisting of `'Antithetic'` and a scalar numeric or logical `1` (`true`) or `0` (`false`).

When you specify `true`, `simByQuadExp` 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.

Note If you specify an input noise process (see `Z`), `simByEuler` ignores the value of `Antithetic`.

Data Types: `logical`

`Z` — Direct specification of dependent random noise process for generating Brownian motion vector

generates correlated Gaussian variates based on the `Correlation` member of the `heston`, `bates`, or `cir` object (default) | function | three-dimensional array of dependent random variates

Direct specification of the dependent random noise process for generating the Brownian motion vector (Wiener process) that drives the simulation, specified as the comma-separated pair consisting of `'Z'` and a function or 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

Data Types: `double` | `function`

`N` — Dependent random counting process for generating number of jumps
random numbers from Poisson distribution with parameter `JumpFreq` from a `bates` object (default) | three-dimensional array | function

Dependent random counting process for generating the number of jumps, specified as the comma-separated pair consisting of `'N'` and a function or an (`NPeriods` □ `NSteps`) -by-`NJumps`-by-`NTrials` three-dimensional array of dependent random variates. If you specify a function, `N` must return an `NJumps`-by-`1` column vector, and you must call it with two inputs: a real-valued scalar observation time t followed by an `NVars`-by-`1` state vector X_t.

Note The `N` name-value pair argument is supported only when you use a `bates` object for the `MDL` input argument.

Data Types: `double` | `function`

`StorePaths` — Flag that indicates how `Paths` is stored and returned
`true` (default) | logical with values `true` or `false`

Flag that indicates how the output array `Paths` is stored and returned, specified as the comma-separated pair consisting of `'StorePaths'` and a scalar numeric or logical `1` (`true`) or `0` (`false`).

If `StorePaths` is `true` (the default value) or is unspecified, `simByQuadExp` returns `Paths` as a three-dimensional time-series array.

If `StorePaths` is `false` (logical `0`), `simByQuadExp` returns `Paths` as an empty matrix.

Data Types: `logical`

`Processes` — Sequence of end-of-period processes or state vector adjustments
`simByQuadExp` makes no adjustments and performs no processing (default) | function | cell array of functions

Sequence of end-of-period processes or state vector adjustments, specified as the comma-separated pair consisting of `'Processes'` and a function or cell array of functions of the form

$$X_t = P(t, X_t)$$

The `simByQuadExp` function runs processing functions at each interpolation time. The functions must accept the current interpolation time t, and the current state vector X_t and return a state vector that can be an adjustment to the input state.

If you specify more than one processing function, `simByQuadExp` 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.

The end-of-period `Processes` argument allows you to terminate a given trial early. At the end of each time step, `simByQuadExp` 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 you to define a `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).

Data Types: `cell` | `function`

Output Arguments

`Paths` — Simulated paths of correlated state variables

array

Simulated paths of correlated state variables for a heston, bates, or cir model, 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 `StorePaths` is set to `false`, `simByQuadExp` returns `Paths` as an empty matrix.

`Times` — Observation times associated with simulated paths

column vector

Observation times for a heston, bates, or cir model associated with the simulated paths, returned as a `(NPeriods + 1)`-by-`1` column vector. Each element of `Times` is associated with the corresponding row of `Paths`.

`Z` — Dependent random variates for generating Brownian motion vector

array

Dependent random variates for a heston, bates, or cir model for generating the Brownian motion vector (Wiener processes) that drive the simulation, returned as an `(NPeriods ⨉ NSteps)`-by-`NBrowns`-by-`NTrials` three-dimensional time-series array.

`N` — Dependent random variates for generating jump counting process vector

array

Dependent random variates for a bates model for generating the jump counting process vector, returned as a `(NPeriods ⨉ NSteps)`-by-`NJumps`-by-`NTrials` three-dimensional time-series array.

More About

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.

Algorithms

Heston

In the Heston stochastic volatility model, the asset value process and volatility process are defined as

$$dS(t) = \gamma(t)S(t)dt + \sqrt{V(t)}S(t)dW_S(t)$$
$$dV(t) = \kappa(\theta - V(t))dt + \sigma\sqrt{V(t)}dW_V(t)$$

Here:

- γ is the continuous risk-free rate.
- θ is a long-term variance level.
- κ is the mean reversion speed for the variance.
- σ is the volatility of volatility.

CIR

You can simulate any vector-valued CIR process of the form

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

Here:

- 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.

Bates

Bates models are bivariate composite models. Each Bates model consists of two coupled univariate models.

- A geometric Brownian motion (`gbm`) model with a stochastic volatility function and jumps that is expressed as follows.

  $$dX_{1t} = B(t)X_{1t}dt + \sqrt{X_{2t}}X_{1t}dW_{1t} + Y(t)X_{1t}dN_t$$

  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 that is expressed as follows.

  $$dX_{2t} = S(t)[L(t) - X_{2t}]dt + V(t)\sqrt{X_{2t}}dW_{2t}$$

  This model describes the evolution of the variance rate of the coupled Bates price process.

Version History

Introduced in R2020a

R2022a: Perform Quasi-Monte Carlo simulation

Behavior changed in R2022a

Perform Quasi-Monte Carlo simulation using the name-value arguments `MonteCarloMethod` and `QuasiSequence`.

R2022b: Perform Brownian bridge and principal components construction

Behavior changed in R2022b

Perform Brownian bridge and principal components construction using the name-value argument `BrownianMotionMethod`.

References

[1] Andersen, Leif. “Simple and Efficient Simulation of the Heston Stochastic Volatility Model.” *The Journal of Computational Finance* 11, no. 3 (March 2008): 1–42.

[2] Broadie, M., and O. Kaya. “Exact Simulation of Option Greeks under Stochastic Volatility and Jump Diffusion Models.” In *Proceedings of the 2004 Winter Simulation Conference*, 2004., 2:535–43. Washington, D.C.: IEEE, 2004.

[3] Broadie, Mark, and Özgür Kaya. “Exact Simulation of Stochastic Volatility and Other Affine Jump Diffusion Processes.” *Operations Research* 54, no. 2 (April 2006): 217–31.

See Also

`bates` | `heston` | `cir` | `simByEuler` | `simByTransition`

Topics

“Simulating Equity Prices” on page 14-28
“Simulating Interest Rates” on page 14-48
“Stratified Sampling” on page 14-57
“Price American Basket Options Using Standard Monte Carlo and Quasi-Monte Carlo Simulation” on page 14-70
“Base SDE Models” on page 14-14
“Drift and Diffusion Models” on page 14-16
“Linear Drift Models” on page 14-19
“Parametric Models” on page 14-21
“SDEs” on page 14-2
“SDE Models” on page 14-7
“SDE Class Hierarchy” on page 14-5
“Quasi-Monte Carlo Simulation” on page 14-62
“Performance Considerations” on page 14-64

simByEuler

Euler simulation of stochastic differential equations (SDEs) for SDE, BM, GBM, CEV, CIR, HWV, Heston, SDEDDO, SDELD, or SDEMRD models

Syntax

```
[Paths,Times,Z] = simByEuler(MDL,NPeriods)
[Paths,Times,Z] = simByEuler( ___ ,Name,Value)
```

Description

[Paths,Times,Z] = simByEuler(MDL,NPeriods) simulates NTrials sample paths of NVars correlated state variables driven by NBrowns Brownian motion sources of risk over NPeriods consecutive observation periods. simByEuler uses the Euler approach to approximate continuous-time stochastic processes.

[Paths,Times,Z] = simByEuler(___ ,Name,Value) specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

You can perform quasi-Monte Carlo simulations using the name-value arguments for MonteCarloMethod, QuasiSequence, and BrownianMotionMethod. For more information, see "Quasi-Monte Carlo Simulation" on page 14-62.

Examples

Simulate Equity Markets Using simByEuler

Load the Data and Specify the SDE Model

```
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);
```

Simulate a Single Path Over a Year

```
nPeriods = 249;      % # of simulated observations
dt       =   1;      % time increment = 1 day
rng(142857,'twister')
[S,T] = simByEuler(SDE, nPeriods, 'DeltaTime', dt);
```

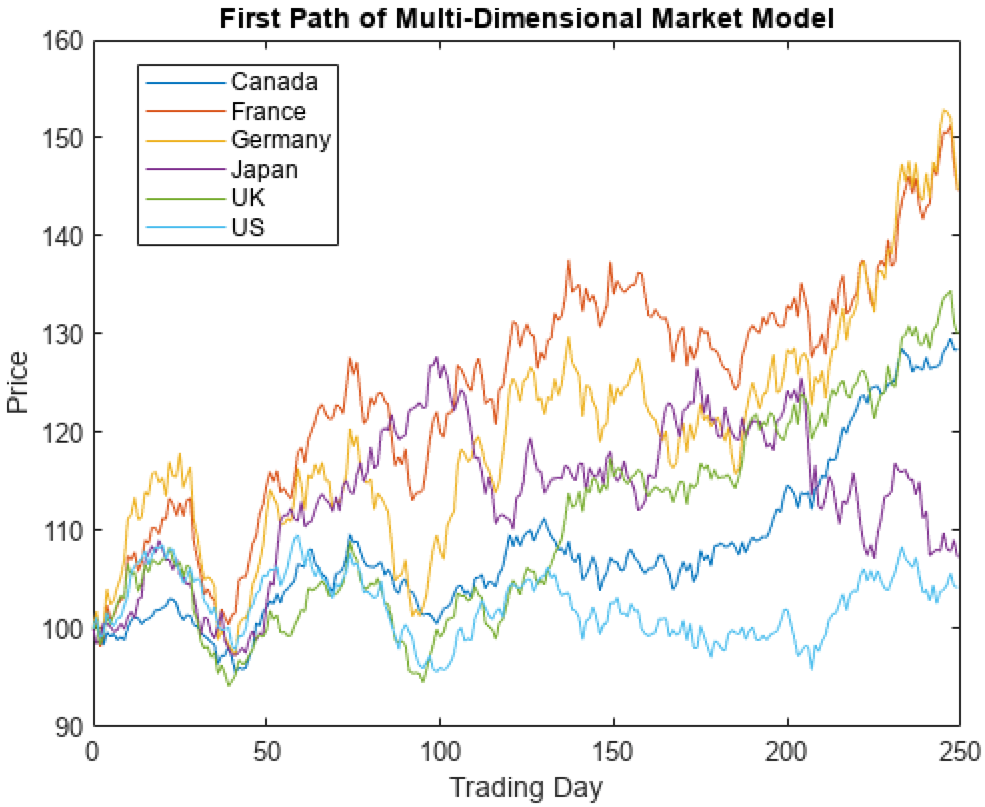
Simulate 10 trials and examine the SDE model

```
rng(142857,'twister')
[S,T] = simulate(SDE, nPeriods, 'DeltaTime', dt, 'nTrials', 10);

whos S
```

```
  Name          Size                 Bytes  Class     Attributes

  S           250x6x10              120000  double
```

Plot the 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')
```



Euler Simulation for 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: $dX_t = 0.2(0.1 - X_t)dt + 0.05X_t^{\frac{1}{2}}dW$.

```
CIR = cir(0.2, 0.1, 0.05)  % (Speed, Level, Sigma)
```

```
CIR = 
   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
```

Simulate a single path over a year using `simByEuler`.

```
nPeriods = 249;      % # of simulated observations
dt       =   1;      % time increment = 1 day
rng(142857,'twister')
[Paths,Times] = simByEuler(CIR,nPeriods,'Method','higham-mao','DeltaTime', dt)
```

```
Paths = 250×1

    1.0000
    0.8613
    0.7245
    0.6349
    0.4741
    0.3853
    0.3374
    0.2549
    0.1859
    0.1814
      ⋮

Times = 250×1

     0
     1
     2
     3
```

```
4
5
6
7
8
9
 ⋮
```

Quasi-Monte Carlo Simulation with `simByEuler` Using a CIR Model

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

```
cir_obj = cir(0.2, 0.1, 0.05)  % (Speed, Level, Sigma)
```

```
cir_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
```

Define the quasi-Monte Carlo simulation using the optional name-value arguments for `'MonteCarloMethod'`,`'QuasiSequence'`, and `'BrownianMotionMethod'`.

```
[paths,time,z] = simByEuler(cir_obj,10,'ntrials',4096,'method','basic','montecarlomethod','quasi
```

Quasi-Monte Carlo Simulation with `simByEuler` Using a Heston Model

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.

```
heston_obj = heston (0.1, 0.2, 0.1, 0.05)  % (Return, Speed, Level, Volatility)
```

```
heston_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
```

Define the quasi-Monte Carlo simulation using the optional name-value arguments for 'MonteCarloMethod','QuasiSequence', and 'BrownianMotionMethod'.

```
[paths,time,z] = simByEuler(heston_obj,10,'ntrials',4096,'montecarlomethod','quasi','quasisequen
```

Quasi-Monte Carlo Simulation with simByEuler Using GBM Model

Create a univariate gbm object to represent the model: $dX_t = 0.25X_tdt + 0.3X_tdW_t$.

```
gbm_obj = gbm(0.25, 0.3)  % (B = Return, Sigma)
```

```
gbm_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.

Define the quasi-Monte Carlo simulation using the optional name-value arguments for 'MonteCarloMethod','QuasiSequence', and 'BrownianMotionMethod'.

```
[paths,time,z] = simByEuler(gbm_obj,10,'ntrials',4096,'montecarlomethod','quasi','quasisequence',
```

Input Arguments

`MDL` — Stochastic differential equation model

object

Stochastic differential equation model, specified as an `sde`, `bm`, `gbm`, `cev`, `cir`, `hwv`, `heston`, `sdeddo`, `sdeld`, or `sdemrd` object.

Data Types: `object`

`NPeriods` — Number of simulation periods

positive scalar integer

Number of simulation periods, specified as a positive scalar integer. The value of `NPeriods` determines the number of rows of the simulated output series.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Paths,Times,Z] = simByEuler(SDE,NPeriods,'DeltaTime',dt)`

`Method` — Method to handle negative values

`'basic'` (default) | character vector with values`'basic'`, `'absorption'`, `'reflection'`, `'partial-truncation'`, `'full-truncation'`, or `'higham-mao'` | string with values `"basic"`, `"absorption"`, `"reflection"`, `"partial-truncation"`, `"full-truncation"`, or `"higham-mao"`

Method to handle negative values, specified as the comma-separated pair consisting of `'Method'` and a character vector or string with a supported value.

Note The `Method` argument is only supported when using a `CIR` object. For more information on creating a `CIR` object, see `cir`.

Data Types: `char` | `string`

`NTrials` — Simulated trials (sample paths) of `NPeriods` observations each

`1` (single path of correlated state variables) (default) | positive scalar integer

Simulated trials (sample paths) of `NPeriods` observations each, specified as the comma-separated pair consisting of `'NTrials'` and a positive scalar integer.

Data Types: `double`

`DeltaTimes` — Positive time increments between observations

`1` (default) | scalar | column vector

Positive time increments between observations, specified as the comma-separated pair consisting of `'DeltaTimes'` and a scalar or a `NPeriods`-by-`1` column vector.

`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.

Data Types: `double`

`NSteps` — Number of intermediate time steps within each time increment *dt* (specified as `DeltaTime`)

`1` (indicating no intermediate evaluation) (default) | positive scalar integer

Number of intermediate time steps within each time increment *dt* (specified as `DeltaTime`), specified as the comma-separated pair consisting of `'NSteps'` and a positive scalar integer.

The `simByEuler` function 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.

Data Types: `double`

`Antithetic` — Flag to indicate whether `simByEuler` uses antithetic sampling to generate the Gaussian random variates

`False` (no antithetic sampling) (default) | logical with values `True` or `False`

Flag indicates whether `simByEuler` uses antithetic sampling to generate the Gaussian random variates that drive the Brownian motion vector (Wiener processes). This argument is specified as the comma-separated pair consisting of `'Antithetic'` and a scalar logical flag with a value of `True` or `False`.

When you specify `True`, `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.

Note If you specify an input noise process (see `Z`), `simByEuler` ignores the value of `Antithetic`.

Data Types: `logical`

`Z` — Direct specification of the dependent random noise process used to generate the Brownian motion vector

generates correlated Gaussian variates based on the `Correlation` member of the `SDE` object (default) | function | three-dimensional array of dependent random variates

Direct specification of the dependent random noise process used to generate the Brownian motion vector (Wiener process) that drives the simulation. This argument is specified as the comma-separated pair consisting of `'Z'` and a function or as an `(NPeriods ⨉ NSteps)`-by-`NBrowns`-by-`NTrials` three-dimensional array of dependent random variates.

Note 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 | function

StorePaths — Flag that indicates how the output array Paths is stored and returned
True (default) | logical with values True or False

Flag that indicates how the output array Paths is stored and returned, specified as the comma-separated pair consisting of 'StorePaths' and a scalar logical flag with a value of True or False.

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.

Data Types: logical

MonteCarloMethod — Monte Carlo method to simulate stochastic processes
"standard" (default) | string with values "standard", "quasi", or "randomized-quasi" | character vector with values 'standard', 'quasi', or 'randomized-quasi'

Monte Carlo method to simulate stochastic processes, specified as the comma-separated pair consisting of 'MonteCarloMethod' and a string or character vector with one of the following values:

- "standard" — Monte Carlo using pseudo random numbers.
- "quasi" — Quasi-Monte Carlo using low-discrepancy sequences.
- "randomized-quasi" — Randomized quasi-Monte Carlo.

Note If you specify an input noise process (see Z), simByEuler ignores the value of MonteCarloMethod.

Data Types: string | char

QuasiSequence — Low discrepancy sequence to drive the stochastic processes
"sobol" (default) | string with value "sobol" | character vector with value 'sobol'

Low discrepancy sequence to drive the stochastic processes, specified as the comma-separated pair consisting of 'QuasiSequence' and a string or character vector with one of the following values:

- "sobol" — Quasi-random low-discrepancy sequences that use a base of two to form successively finer uniform partitions of the unit interval and then reorder the coordinates in each dimension

Note If MonteCarloMethod option is not specified or specified as"standard", QuasiSequence is ignored.

Data Types: string | char

BrownianMotionMethod — Brownian motion construction method

"standard" (default) | string with value "brownian-bridge" or "principal-components" | character vector with value 'brownian-bridge' or 'principal-components'

Brownian motion construction method, specified as the comma-separated pair consisting of 'BrownianMotionMethod' and a string or character vector with one of the following values:

- "standard" — The Brownian motion path is found by taking the cumulative sum of the Gaussian variates.
- "brownian-bridge" — The last step of the Brownian motion path is calculated first, followed by any order between steps until all steps have been determined.
- "principal-components" — The Brownian motion path is calculated by minimizing the approximation error.

Note If an input noise process is specified using the Z input argument, BrownianMotionMethod is ignored.

The starting point for a Monte Carlo simulation is the construction of a Brownian motion sample path (or Wiener path). Such paths are built from a set of independent Gaussian variates, using either standard discretization, Brownian-bridge construction, or principal components construction.

Both standard discretization and Brownian-bridge construction share the same variance and therefore the same resulting convergence when used with the MonteCarloMethod using pseudo random numbers. However, the performance differs between the two when the MonteCarloMethod option "quasi" is introduced, with faster convergence seen for "brownian-bridge" construction option and the fastest convergence when using the "principal-components" construction option.

Data Types: string | char

Processes — Sequence of end-of-period processes or state vector adjustments

simByEuler makes no adjustments and performs no processing (default) | function | cell array of functions

Sequence of end-of-period processes or state vector adjustments of the form, specified as the comma-separated pair consisting of 'Processes' and a function or cell array of functions of the form

$$X_t = P(t, X_t)$$

The simByEuler function 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, 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.

Data Types: cell | function

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`, `simByEuler` returns `Paths` as an empty matrix.

`Times` — Observation times associated with the simulated paths
column vector

Observation times associated with the simulated paths, returned as a `(NPeriods + 1)`-by-`1` column vector. Each element of `Times` is associated with the corresponding row of `Paths`.

`Z` — Dependent random variates used to generate the Brownian motion vector
array

Dependent random variates used to generate the Brownian motion vector (Wiener processes) that drive the simulation, returned as a `(NPeriods □ NSteps)`-by-`NBrowns`-by-`NTrials` three-dimensional time series array.

More About

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.

Algorithms

This function 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.

- 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).

Version History

Introduced in R2008a

R2022a: Perform Quasi-Monte Carlo simulation

Behavior changed in R2022a

Perform Quasi-Monte Carlo simulation using the name-value arguments MonteCarloMethod and QuasiSequence.

R2022b: Perform Brownian bridge and principal components construction

Behavior changed in R2022b

Perform Brownian bridge and principal components construction using the name-value argument BrownianMotionMethod.

References

[1] Deelstra, G. and F. Delbaen. “Convergence of Discretized Stochastic (Interest Rate) Processes with Stochastic Drift Term.” *Applied Stochastic Models and Data Analysis.*, 1998, vol. 14, no. 1, pp. 77–84.

[2] Higham, Desmond, and Xuerong Mao. “Convergence of Monte Carlo Simulations Involving the Mean-Reverting Square Root Process.” *The Journal of Computational Finance*, vol. 8, no. 3, 2005, pp. 35–61.

[3] Lord, Roger, et al. “A Comparison of Biased Simulation Schemes for Stochastic Volatility Models.” *Quantitative Finance*, vol. 10, no. 2, Feb. 2010, pp. 177–94

See Also

simByTransition | simBySolution | simBySolution | simulate | sde | bm | gbm | sdeddo | sdeld | cev | cir | heston | hwv | sdemrd

Topics

Implementing Multidimensional Equity Market Models, Implementation 5: Using the simByEuler Method on page 14-33
“Simulating Equity Prices” on page 14-28
“Simulating Interest Rates” on page 14-48
“Stratified Sampling” on page 14-57
“Price American Basket Options Using Standard Monte Carlo and Quasi-Monte Carlo Simulation” on page 14-70
“Base SDE Models” on page 14-14
“Drift and Diffusion Models” on page 14-16
“Linear Drift Models” on page 14-19
“Parametric Models” on page 14-21
“SDEs” on page 14-2
“SDE Models” on page 14-7
“SDE Class Hierarchy” on page 14-5
“Quasi-Monte Carlo Simulation” on page 14-62
“Performance Considerations” on page 14-64

simByTransition

Simulate `CIR` sample paths with transition density

Syntax

```
[Paths,Times] = simByTransition(MDL,NPeriods)
[Paths,Times] = simByTransition(___,Name,Value)
```

Description

`[Paths,Times] = simByTransition(MDL,NPeriods)` simulates `NTrials` sample paths of `NVars` independent state variables driven by the Cox-Ingersoll-Ross (CIR) process sources of risk over `NPeriods` consecutive observation periods. `simByTransition` approximates a continuous-time CIR model using an approximation of the transition density function.

`[Paths,Times] = simByTransition(___,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

You can perform quasi-Monte Carlo simulations using the name-value arguments for `MonteCarloMethod`, `QuasiSequence`, and `BrownianMotionMethod`. For more information, see "Quasi-Monte Carlo Simulation" on page 14-62.

Examples

Simulate Future Term Structures Using a CIR Model

Using the short rate, simulate the rate dynamics and term structures in the future using a CIR model. The CIR model is expressed as

$$dr(t) = \alpha(b - r(t))dt + \sigma\sqrt{r(t)}dW(t)$$

The exponential affine form of the bond price is

$$B(t,T) = e^{-A(t,T)r(t) + C(t,T)}$$

where

$$A(t,T) = \frac{2(e^{\gamma(T-t)} - 1)}{(\gamma + \alpha)(e^{\gamma(T-t)} - 1) + 2\gamma}$$

$$B(t,T) = \frac{2\alpha b}{\sigma^2}\log\left(\frac{2\gamma e^{(\alpha+\gamma)(T-t)/2}}{(\gamma + \alpha)(e^{\gamma(T-t)} - 1) + 2\gamma}\right)$$

and

$$\gamma = \sqrt{\alpha^2 + 2\sigma^2}$$

Define the parameters for the `cir` object.

```
alpha = .1;
b = .05;
sigma = .05;
r0 = .04;
```

Define the function for bond prices.

```
gamma = sqrt(alpha^2 + 2*sigma^2);
A_func = @(t, T) ...
    2*(exp(gamma*(T-t))-1)/((alpha+gamma)*(exp(gamma*(T-t))-1)+2*gamma);
C_func = @(t, T) ...
    (2*alpha*b/sigma^2)*log(2*gamma*exp((alpha+gamma)*(T-t)/2)/((alpha+gamma)*(exp(gamma*(T-t))-1
P_func = @(t,T,r_t) exp(-A_func(t,T)*r_t+C_func(t,T));
```

Create a `cir` object.

```
obj = cir(alpha,b,sigma,'StartState',r0)
```

```
obj = 
   Class CIR: Cox-Ingersoll-Ross
   ----------------------------------------
     Dimensions: State = 1, Brownian = 1
   ----------------------------------------
      StartTime: 0
     StartState: 0.04
    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: 0.05
          Speed: 0.1
```

Define the simulation parameters.

```
nTrials = 100;
nPeriods = 5;   % Simulate future short over the next five years
nSteps = 12;    % Set intermediate steps to improve the accuracy
```

Simulate the short rates. The returning path is a (NPeriods + 1)-by-`NVars`-by-`NTrials` three-dimensional time-series array. For this example, the size of the output is `6`-by-`1`-by-`100`.

```
rng('default');    % Reproduce the same result
rPaths = simByTransition(obj,nPeriods,'nTrials',nTrials,'nSteps',nSteps);
size(rPaths)
```

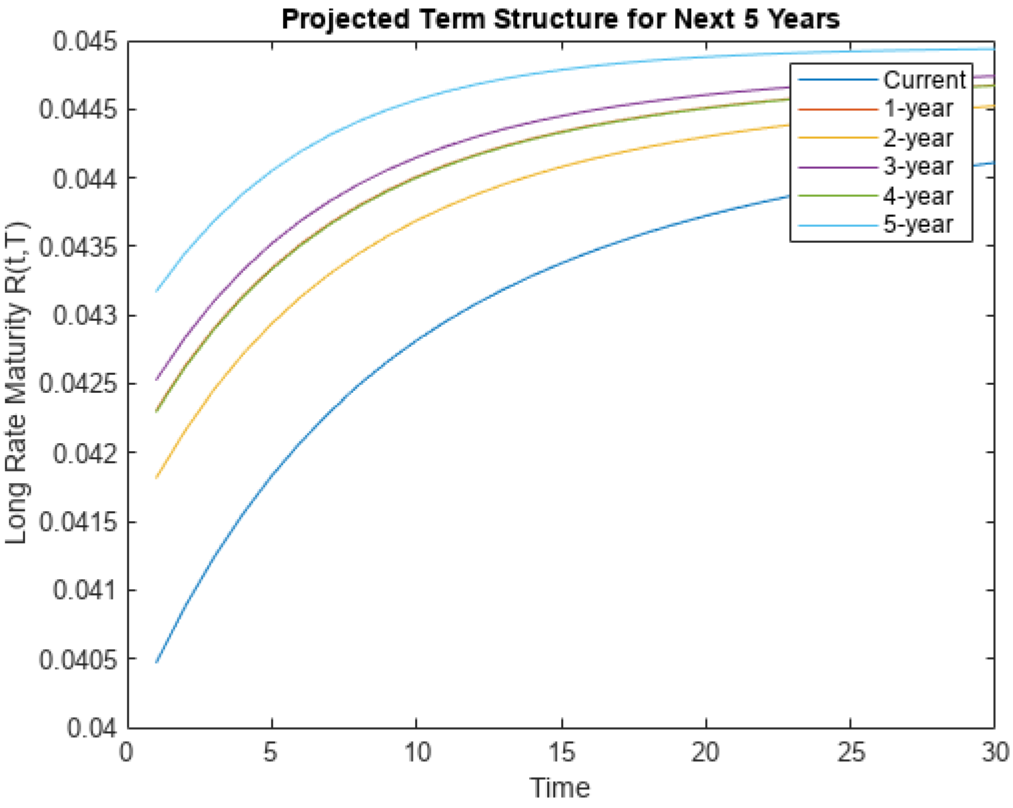
```
ans = 1×3

     6     1   100

```

```
rPathsExp = mean(rPaths,3);
```

Determine the term structure over the next 30 years.

```
maturity = 30;
T = 1:maturity;
futuresTimes = 1:nPeriods+1;
```

```
% Preallocate simTermStruc
simTermStructure = zeros(nPeriods+1,30);
for i = futuresTimes
    for t = T
        bondPrice = P_func(i,i+t,rPathsExp(i));
        simTermStructure(i,t) = -log(bondPrice)/t;
    end
end
plot(simTermStructure')
legend('Current','1-year','2-year','3-year','4-year','5-year')
title('Projected Term Structure for Next 5 Years')
ylabel('Long Rate Maturity R(t,T)')
xlabel('Time')
```



Quasi-Monte Carlo Simulation Using a CIR Model

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

```
cir_obj = cir(0.2, 0.1, 0.05)  % (Speed, Level, Sigma)
```

```
cir_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
```

Define the quasi-Monte Carlo simulation using the optional name-value arguments for `'MonteCarloMethod'`,`'QuasiSequence'`, and `'BrownianMotionMethod'`.

```
[paths,time] = simByTransition(cir_obj,10,'ntrials',4096,'montecarlomethod','quasi','quasisequen
```

Input Arguments

`MDL` — Stochastic differential equation model

object

Stochastic differential equation model, specified as a `cir` object. For more information on creating a `CIR` object, see `cir`.

Data Types: `object`

`NPeriods` — Number of simulation periods

positive scalar integer

Number of simulation periods, specified as a positive scalar integer. The value of `NPeriods` determines the number of rows of the simulated output series.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Paths,Times] = simByTransition(CIR,NPeriods,'DeltaTimes',dt)`

`NTrials` — Simulated trials (sample paths)

`1` (single path of correlated state variables) (default) | positive integer

Simulated trials (sample paths) of NPeriods observations each, specified as the comma-separated pair consisting of 'NTrials' and a positive scalar integer.

Data Types: double

DeltaTimes — Positive time increments between observations
1 (default) | scalar | column vector

Positive time increments between observations, specified as the comma-separated pair consisting of 'DeltaTimes' and a scalar or a NPeriods-by-1 column vector.

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.

Data Types: double

NSteps — Number of intermediate time steps
1 (indicating no intermediate evaluation) (default) | positive integer

Number of intermediate time steps within each time increment *dt* (defined as DeltaTimes), specified as the comma-separated pair consisting of 'NSteps' and a positive scalar integer.

The simByTransition function 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 simByTransition does not report the output state vector at these intermediate points, the refinement improves accuracy by enabling the simulation to more closely approximate the underlying continuous-time process.

Data Types: double

StorePaths — Flag for storage and return method
True (default) | logical with values True or False

Flag for storage and return method that indicates how the output array Paths is stored and returned, specified as the comma-separated pair consisting of 'StorePaths' and a scalar logical flag with a value of True or False.

- If StorePaths is True (the default value) or is unspecified, then simByTransition returns Paths as a three-dimensional time series array.
- If StorePaths is False (logical 0), then simByTransition returns the Paths output array as an empty matrix.

Data Types: logical

MonteCarloMethod — Monte Carlo method to simulate stochastic processes
"standard" (default) | string with values "standard", "quasi", or "randomized-quasi" | character vector with values 'standard', 'quasi', or 'randomized-quasi'

Monte Carlo method to simulate stochastic processes, specified as the comma-separated pair consisting of 'MonteCarloMethod' and a string or character vector with one of the following values:

- "standard" — Monte Carlo using pseudo random numbers that has a convergence rate of $O(N^{-½})$.
- "quasi" — Quasi-Monte Carlo rate of convergence is faster than standard Monte Carlo with errors approaching size of $O(N^{-1})$.

- "randomized-quasi" — Quasi-random sequences, also called low-discrepancy sequences, are deterministic uniformly distributed sequences which are specifically designed to place sample points as uniformly as possible.

Data Types: string | char

QuasiSequence — Low discrepancy sequence to drive the stochastic processes
"sobol" (default) | string with value "sobol" | character vector with value 'sobol'

Low discrepancy sequence to drive the stochastic processes, specified as the comma-separated pair consisting of 'QuasiSequence' and a string or character vector with one of the following values:

- "sobol" — Quasi-random low-discrepancy sequences that use a base of two to form successively finer uniform partitions of the unit interval and then reorder the coordinates in each dimension

Note If MonteCarloMethod option is not specified or specified as"standard", QuasiSequence is ignored.

Data Types: string | char

BrownianMotionMethod — Brownian motion construction method
"standard" (default) | string with value "brownian-bridge" or "principal-components" | character vector with value 'brownian-bridge' or 'principal-components'

Brownian motion construction method, specified as the comma-separated pair consisting of 'BrownianMotionMethod' and a string or character vector with one of the following values:

- "standard" — The Brownian motion path is found by taking the cumulative sum of the Gaussian variates.
- "brownian-bridge" — The last step of the Brownian motion path is calculated first, followed by any order between steps until all steps have been determined.
- "principal-components" — The Brownian motion path is calculated by minimizing the approximation error.

The starting point for a Monte Carlo simulation is the construction of a Brownian motion sample path (or Wiener path). Such paths are built from a set of independent Gaussian variates, using either standard discretization, Brownian-bridge construction, or principal components construction.

Both standard discretization and Brownian-bridge construction share the same variance and therefore the same resulting convergence when used with the MonteCarloMethod using pseudo random numbers. However, the performance differs between the two when the MonteCarloMethod option "quasi" is introduced, with faster convergence seen for "brownian-bridge" construction option and the fastest convergence when using the "principal-components" construction option.

Data Types: string | char

Processes — Sequence of end-of-period processes or state vector adjustments
simByEuler makes no adjustments and performs no processing (default) | function | cell array of functions

Sequence of end-of-period processes or state vector adjustments, specified as the comma-separated pair consisting of 'Processes' and a function or cell array of functions of the form

$$X_t = P(t, X_t)$$

.

`simByTransition` applies processing functions at the end of each observation period. The processing functions accept the current observation time t and the current state vector X_t, and return a state vector that may adjust the input state.

If you specify more than one processing function, `simByTransition` invokes the functions in the order in which they appear in the cell array.

Data Types: `cell` | `function`

Output Arguments

`Paths` — Simulated paths of correlated state variables

array

Simulated paths of correlated state variables, returned as an `(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`, `simByTransition` returns `Paths` as an empty matrix.

`Times` — Observation times associated with the simulated paths

column vector

Observation times associated with the simulated paths, returned as an `(NPeriods + 1)`-by-1 column vector. Each element of `Times` is associated with the corresponding row of `Paths`.

More About

Transition Density Simulation

The SDE has no solution such that $r(t) = f(r(0),\cdots)$.

In other words, the equation is not explicitly solvable. However, the transition density for the process is known.

The exact simulation for the distribution of $r(t\_1\ ),\cdots,r(t\_n)$ is that of the process at times $t\_1,\cdots,t\_n$ for the same value of $r(0)$. The transition density for this process is known and is expressed as

$$r(t) = \frac{\sigma^2(1 - e^{-\alpha(t-u)}}{4\alpha} x_d^2 \left(\frac{4\alpha e^{-\alpha(t-u)}}{\sigma^2(1 - e^{-\alpha(t-u)})} r(u) \right), t > u$$

where

$$d \equiv \frac{4b\alpha}{\sigma^2}$$

Algorithms

Use the `simByTransition` function to simulate any vector-valued CIR process 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.

Version History

Introduced in R2018b

R2022a: Perform Quasi-Monte Carlo simulation

Behavior changed in R2022a

Perform Quasi-Monte Carlo simulation using the name-value arguments `MonteCarloMethod` and `QuasiSequence`.

R2022b: Perform Brownian bridge and principal components construction

Behavior changed in R2022b

Perform Brownian bridge and principal components construction using the name-value argument `BrownianMotionMethod`.

References

[1] Glasserman, P. *Monte Carlo Methods in Financial Engineering.* New York: Springer-Verlag, 2004.

See Also

`cir` | `simByEuler`

Topics

“SDEs” on page 14-2
“SDE Models” on page 14-7
“SDE Class Hierarchy” on page 14-5
“Quasi-Monte Carlo Simulation” on page 14-62

simulate

Simulate multivariate stochastic differential equations (SDEs) for `SDE`, `BM`, `GBM`, `CEV`, `CIR`, `HWV`, `Heston`, `SDEDDO`, `SDELD`, `SDEMRD`, `Merton`, or `Bates` models

Syntax

```
[Paths,Times,Z] = simulate(MDL)
[Paths,Times,Z] = simulate( ___ ,Optional,Scheme)
```

Description

`[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.

`[Paths,Times,Z] = simulate( ___ ,Optional,Scheme)` adds optional inputs for `Optional` and `Scheme`.

The `Optional` input argument for `simulate` 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.

The optional input `Scheme` lets you specify an approximation scheme used to simulate the sample paths and you can use this optional input with or without an `Optional` input argument.

Examples

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.

Create a GBM model using `gbm`.

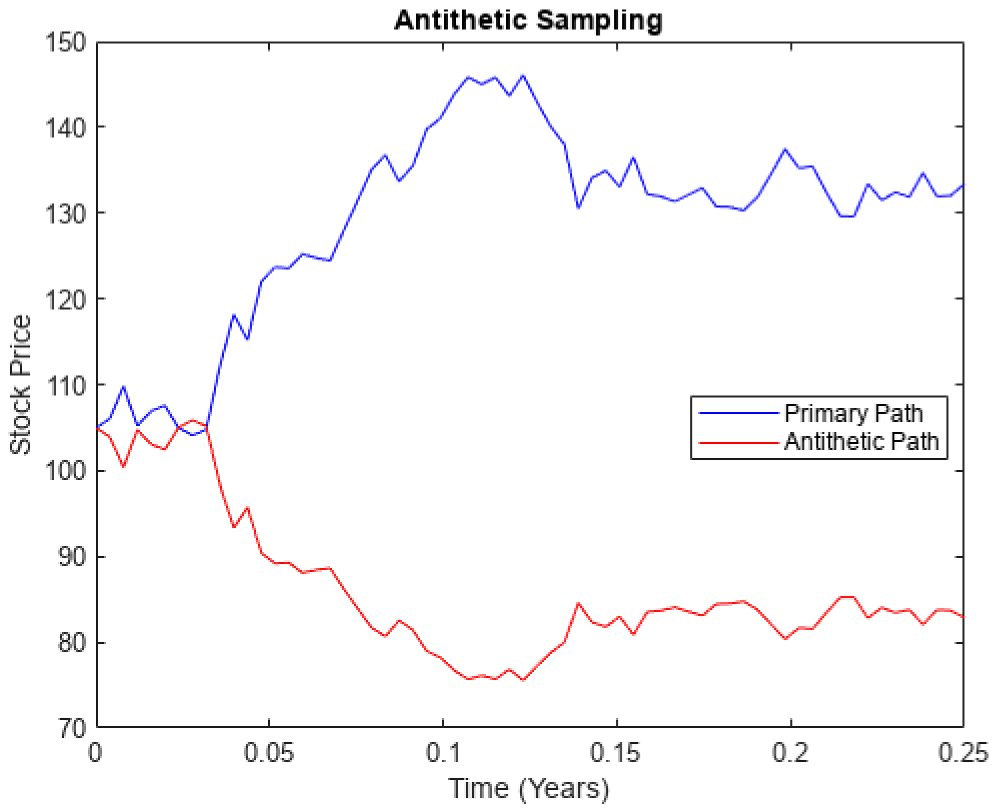
```
barrier      = 120;          % barrier
strike       = 100;          % exercise price
rate         = 0.05;         % annualized risk-free rate
sigma        = 0.3;          % annualized volatility
nPeriods     = 63;           % 63 trading days
dt           = 1 / 252;      % time increment = 252 days
Time        = nPeriods * dt; % expiration time = 0.25 years
obj        = gbm(rate, sigma, 'StartState', 105);
```

Perform a small-scale simulation that explicitly returns two simulated paths.

```
rng('default')               % make output reproducible
[X, T] = obj.simBySolution(nPeriods, 'DeltaTime', dt, ...
                'nTrials', 2, 'Antithetic', true);
```

Perform antithetic sampling such that all primary and antithetic paths are simulated and stored in successive matching pairs. Odd paths (1,3,5,...) correspond to the primary Gaussian paths. Even paths (2,4,6,...) are the matching antithetic paths of each pair, derived by negating the Gaussian draws of the corresponding primary (odd) path. Verify this by examining the matching paths of the primary/antithetic pair.

```
plot(T, X(:,:,1), 'blue', T, X(:,:,2), 'red')
xlabel('Time (Years)'), ylabel('Stock Price'), ...
            title('Antithetic Sampling')
legend({'Primary Path' 'Antithetic Path'}, ...
            'Location', 'Best')
```



To price the European barrier option, specify an end-of-period processing function to record the maximum and terminal stock prices. This processing function is accessible by time and state, and is implemented as a nested function with access to shared information that allows the option price and corresponding standard error to be calculated. For more information on using an end-of-period processing function, see “Pricing Equity Options” on page 14-45.

Simulate 200 paths using the processing function method.

```
rng('default')              % make output reproducible
barrier  = 120;             % barrier
strike   = 100;             % exercise price
rate     = 0.05;            % annualized risk-free rate
sigma    = 0.3;             % annualized volatility
nPeriods = 63;              % 63 trading days
dt       = 1 / 252;         % time increment = 252 days
Time     = nPeriods * dt;   % expiration time = 0.25 years
obj    = gbm(rate, sigma, 'StartState', 105);
nPaths = 200;          % # of paths = 100 sets of pairs
f      = Example_BarrierOption(nPeriods, nPaths);
simulate(obj, nPeriods, 'DeltaTime' , dt, ...
            'nTrials', nPaths, 'Antithetic', true, ...
            'Processes', f.SaveMaxLast);
```

Approximate the option price with a 95% confidence interval.

```
optionPrice   = f.OptionPrice(strike, rate, barrier);
standardError = f.StandardError(strike, rate, barrier,...
                          true);
lowerBound    = optionPrice - 1.96 * standardError;
upperBound    = optionPrice + 1.96 * standardError;

displaySummary(optionPrice, standardError, lowerBound, upperBound);
```

```
  Up-and-In Barrier Option Price:   6.6572
         Standard Error of Price:   0.7292
 Confidence Interval Lower Bound:   5.2280
 Confidence Interval Upper Bound:   8.0864
```

Utility Function

```
function displaySummary(optionPrice, standardError, lowerBound, upperBound)
fprintf('  Up-and-In Barrier Option Price: %8.4f\n', ...
            optionPrice);
fprintf('         Standard Error of Price: %8.4f\n', ...
            standardError);
fprintf(' Confidence Interval Lower Bound: %8.4f\n', ...
            lowerBound);
fprintf(' Confidence Interval Upper Bound: %8.4f\n', ...
            upperBound);
end
```

Specify Approximation Scheme with `simulate` Using a CIR Model

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

```
cir_obj = cir(0.2, 0.1, 0.05)  % (Speed, Level, Sigma)
```

```
cir_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
```

Use the optional name-value argument for `'Scheme'` to specify a scheme to simulate the sample paths.

```
[paths,times] = simulate(cir_obj,10,'ntrials',4096,'scheme','quadratic-exponential');
```

Use simulate with Quasi-Monte Carlo Simulation with simByEuler Using a CIR Model

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

```
cir_obj = cir(0.2, 0.1, 0.05)  % (Speed, Level, Sigma)
```

```
cir_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
```

Use optional name-value inputs for the `simByEuler` method that you can call through `simulate` interface using the `simulate` `'Scheme'` for `'quadratic-exponential'`. The optional inputs for `simByEuler` define a quasi-Monte Carlo simulation using the name-value arguments for `'MonteCarloMethod'`,`'QuasiSequence'`, and `'BrownianMotionMethod'`.

```
[paths,time] = simulate(cir_obj,10,'ntrials',4096,'montecarlomethod','quasi','quasisequence','sol
```

Input Arguments

MDL — Stochastic differential equation model
object

Stochastic differential equation model, specified as an `sde`, `bates`, `bm`, `gbm`, `cev`, `cir`, `hwv`, `heston`,`merton` `sdeddo`, `sdeld`, or `sdemrd` object.

Data Types: `object`

Optional — Any variable-length list of input arguments that the simulation method or function referenced by the SDE.Simulation parameter requires or accepts
input arguments

(Optional) Any variable-length list of input arguments that the simulation method or function referenced by the `SDE.Simulation` parameter requires or accepts, specified as a variable-length list of input arguments. This input list is passed directly to the appropriate SDE simulation method or user-defined simulation function.

Data Types: `double`

`Scheme` — Approximation scheme used to simulate the sample paths
character vector with values `'euler'`, `'solution'`,`'quadratic-exponential'`, `'transition'`, or `'milstein'` | string with values `"euler"`, `"solution"`,`"quadratic-exponential"`, `"transition"`, or `"milstein"`

(Optional) Approximation scheme used to simulate the sample paths, specified as a character vector or string. When you specify `Scheme`, the MDL simulation method using the corresponding scheme is used instead of the one defined in the `MDL.Simulation` property which is the `simByEuler` method by default.

Note The supported `Scheme` depends on which SDE object you use with the `simulate` method as follows:

`simulate` Scheme Value	Supported Objects
`"euler"`	`sde`, `sdeddo`, `sdeld`, `sdemrd`, `bm`, `cev`, `gbm`, `merton`, `hwv`, `cir`, `heston`, or `bates`
`"solution"`	`gbm`, `merton`, or `hwv`
`"transition"`	`cir`, `heston`, or `bates`
`"quadratic-exponential"`	`cir`, `heston`, or `bates`
`"milstein"`	`sdeddo`, `sdeld`, `sdemrd`, `bm`, `cev`, `gbm`, `merton`, `hwv`, `cir`, `heston`, or `bates`

Data Types: `char` | `string`

Output Arguments

`Paths` — Three-dimensional time series array, consisting of simulated paths of correlated state variables
array

Three-dimensional time series array, consisting of simulated paths of correlated state variables, returned as an `(NPeriods + 1)`-by-`NVars`-by-`NTrials` array.

For a given trial, each row of `Paths` is the transpose of the state vector X_t at time t.

`Times` — Observation times associated with the simulated paths
column vector

Observation times associated with the simulated paths, returned as a `(NPeriods + 1)`-by-`1` column vector.

Z — Three-dimensional time series array of dependent random variates used to generate the Brownian motion vector (Wiener processes)
array | matrix | table | timetable

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`, returned as a `NTimes`-by-`NBrowns`-by-`NTrials` array.

`NTimes` is the number of time steps at which the `simulate` function samples the state vector. `NTimes` includes intermediate times designed to improve accuracy, which `simulate` does not necessarily report in the `Paths` output time series.

More About

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.

Algorithms

This function simulates any vector-valued SDE of the form:

$$dX_t = F(t, X_t)dt + G(t, X_t)dW_t \qquad (15\text{-}4)$$

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.

Version History

Introduced in R2008a

R2023a: Added "milstein" value to Scheme name-value argument
Behavior changed in R2023a

When the MDL input argument is a cir, heston, bates, or merton object, you can use a Scheme value of "milstein".

Use the simByMilstein method to approximate a numerical solution of a stochastic differential equation.

R2022b: Approximation scheme to simulate the sample paths
Behavior changed in R2022b

When you specify Scheme, the MDL simulation method using the corresponding scheme is used instead of the one defined in the MDL.Simulation property.

References

[1] Ait-Sahalia, Y. “Testing Continuous-Time Models of the Spot Interest Rate.” *The Review of Financial Studies*, Spring 1996, Vol. 9, No. 2, pp. 385–426.

[2] Ait-Sahalia, Y. “Transition Densities for Interest Rate and Other Nonlinear Diffusions.” *The Journal of Finance*, Vol. 54, No. 4, August 1999.

[3] Glasserman, P. *Monte Carlo Methods in Financial Engineering.* New York, Springer-Verlag, 2004.

[4] Hull, J. C. *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

[5] Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions.* Vol. 2, 2nd ed. New York, John Wiley & Sons, 1995.

[6] Shreve, S. E. *Stochastic Calculus for Finance II: Continuous-Time Models.* New York: Springer-Verlag, 2004.

See Also

simByEuler | simBySolution | simBySolution | sde | merton | bates | bm | gbm | sdeddo | sdeld | cev | cir | heston | hwv | sdemrd

Topics
“Simulating Equity Prices” on page 14-28
“Simulating Interest Rates” on page 14-48
“Stratified Sampling” on page 14-57
“Price American Basket Options Using Standard Monte Carlo and Quasi-Monte Carlo Simulation” on page 14-70
“Base SDE Models” on page 14-14
“Drift and Diffusion Models” on page 14-16
“Linear Drift Models” on page 14-19
“Parametric Models” on page 14-21
“SDEs” on page 14-2
“SDE Models” on page 14-7
“SDE Class Hierarchy” on page 14-5
“Performance Considerations” on page 14-64

ts2func

Convert time series arrays to functions of time and state

Syntax

```
F = ts2func(Array)
F = ts2func(___,Name,Value)
```

Description

`F = ts2func(Array)` 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.

n periods.

`F = ts2func(___,Name,Value)` adds optional name-value pair arguments.

Examples

Illustrate the Dynamic Behavior of Market Parameters

Load the data.

```
load Data_GlobalIdx2
```

Simulate risk-neutral sample paths.

```
dt      = 1/250;
returns = tick2ret(Dataset.CAC);
sigma   = std(returns)*sqrt(250);
yields  = Dataset.EB3M;
yields  = 360*log(1 + yields);
```

Simulate paths using a constant, 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);
```

Simulate paths using a dynamic, deterministic rate of return (get r)

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

Simulate paths using a dynamic, deterministic rate of return (r output 1)

```
r(0,100)
```

```
ans = 0.0470
```

Simulate paths using a dynamic, deterministic rate of return (r output 2)

```
r(7.5,200)
```

```
ans = 0.0472
```

Simulate paths using a dynamic, deterministic rate of return (r output 3)

```
r(7.5)
```

```
ans = 0.0472
```

Simulate paths using a dynamic, deterministic rate of return

```
rng(5713,'twister')
obj = gbm(r, diag(sigma),'StartState',100)
```

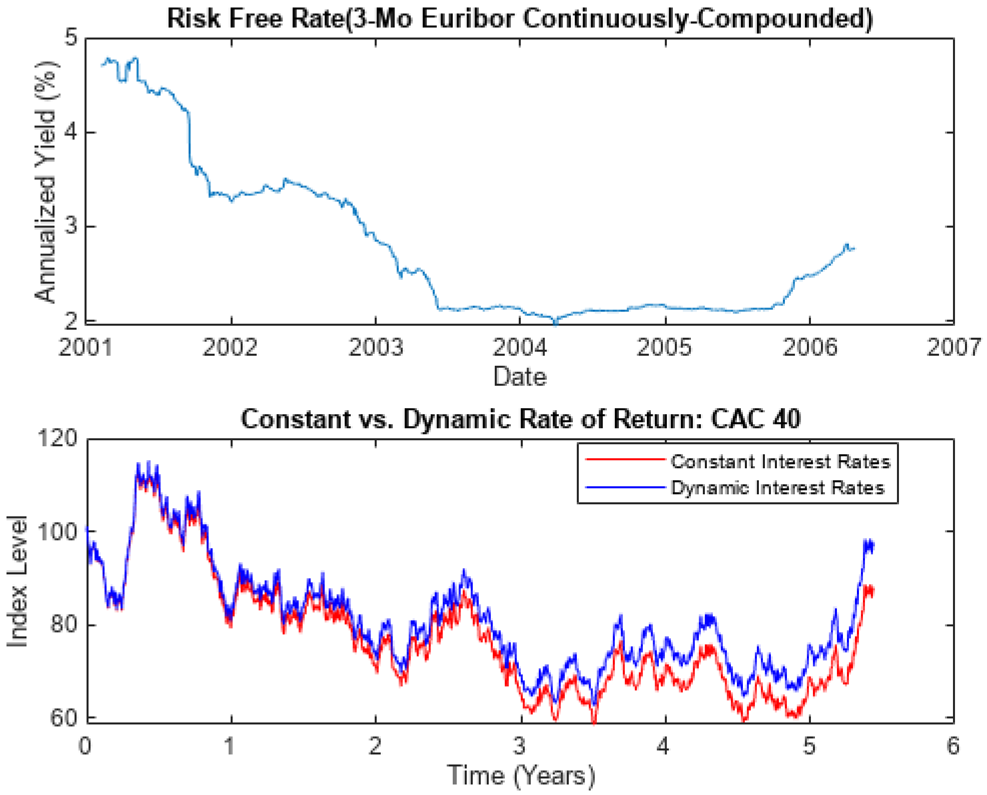
```
obj = 
   Class GBM: Generalized Geometric Brownian Motion
   ------------------------------------------------
     Dimensions: State = 1, Brownian = 1
   ------------------------------------------------
      StartTime: 0
     StartState: 100
    Correlation: 1
          Drift: drift rate function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

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')
```



Input Arguments

`Array` — Time series array to encapsulate within a callable function of time and state

vector | 2-dimensional matrix | three-dimensional array

Time series array to encapsulate within a callable function of time and state, specified as a vector, two-dimensional matrix, or three-dimensional array

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `F = ts2func(yields,'Times',(0:nPeriods - 1)')`

Times — Monotonically increasing observation times associated with the time series input array Array

zero-based, unit-increment vector of the same length as that of the dimension of Array (default) | vector

Monotonically increasing observation times associated with the time series input array (Array), specified as the comma-separated pair consisting of 'Times' and a vector.

Data Types: double

TimeDimension — Specifies which dimension of the input time series array Array is associated with time

1 (indicates that time is associated with the rows of Array) (default) | scalar integer

Specifies which dimension of the input time series array (Array) is associated with time, specified as the comma-separated pair consisting of 'TimeDimension' and a scalar integer.

Data Types: double

StateDimension — Specifies which dimension of the input time series array Array is associated with the NVars state variables

first dimension of Array not already associated with time is associated with first available dimension of Array not already assigned to TimeDimension (default) | positive scalar integer

Specifies which dimension of the input time series array (Array) is associated with the NVars state variables, specified as the comma-separated pair consisting of 'StateDimension' and a positive scalar integer.

Data Types: double

Deterministic — Flag to indicate whether the output function is a deterministic function of time alone

false (meaning F is a callable function of time and state, *F(t,X)*) (default) | scalar logical

Flag to indicate whether the output function is a deterministic function of time alone, specified as the comma-separated pair consisting of 'Determininistic' and a scalar integer flag.

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)*.

Data Types: logical

Output Arguments

F — Callable function *F(t)* of a real-valued scalar observation time *t*

vector | matrix | table | timetable

Callable function *F(t)* of a real-valued scalar observation time *t*, returned as a function.

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)*.

Note 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.

Algorithms

- When you specify `Array` as a scalar or a vector (row or column), `ts2func` assumes that it represents a univariate time series.
- `F` returns an array with one less 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 two-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.

Version History

Introduced in R2008a

References

[1] Ait-Sahalia, Y. “Testing Continuous-Time Models of the Spot Interest Rate.” *The Review of Financial Studies*, Spring 1996, Vol. 9, No. 2, pp. 385–426.

[2] Ait-Sahalia, Y. “Transition Densities for Interest Rate and Other Nonlinear Diffusions.” *The Journal of Finance*, Vol. 54, No. 4, August 1999.

[3] Glasserman, P. *Monte Carlo Methods in Financial Engineering.* New York, Springer-Verlag, 2004.

[4] Hull, J. C. *Options, Futures, and Other Derivatives*, 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

[5] Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions.* Vol. 2, 2nd ed. New York, John Wiley & Sons, 1995.

[6] 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 14-28
“Simulating Interest Rates” on page 14-48
“Stratified Sampling” on page 14-57
“Price American Basket Options Using Standard Monte Carlo and Quasi-Monte Carlo Simulation” on page 14-70
“Base SDE Models” on page 14-14
“Drift and Diffusion Models” on page 14-16
“Linear Drift Models” on page 14-19
“Parametric Models” on page 14-21
“SDEs” on page 14-2
“SDE Models” on page 14-7
“SDE Class Hierarchy” on page 14-5
“Performance Considerations” on page 14-64

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).

Examples

Convert Constraints in Terms of Absolute Weights Into Constraints in Terms of Active Portfolio Weights

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:

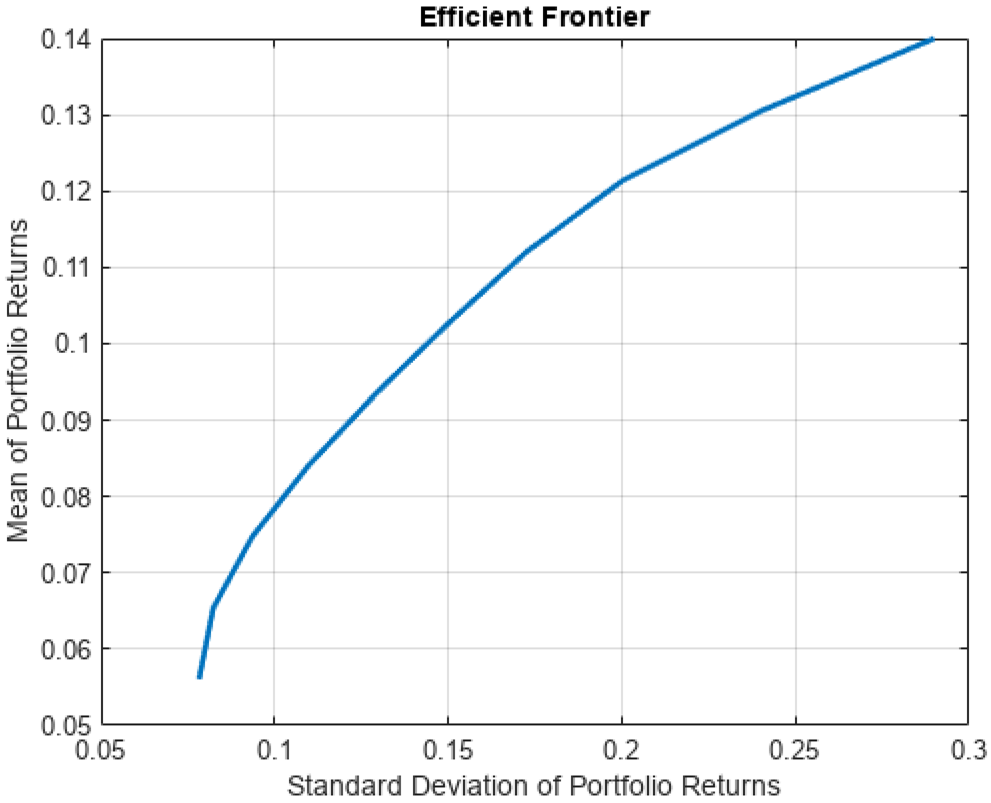
```
m = [ 0.14; 0.10; 0.05 ];
C = [ 0.29^2 0.4*0.29*0.17 0.1*0.29*0.08; 0.4*0.29*0.17 0.17^2 0.3*0.17*0.08;...
0.1*0.29*0.08 0.3*0.17*0.08 0.08^2 ];
```

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 ];
```

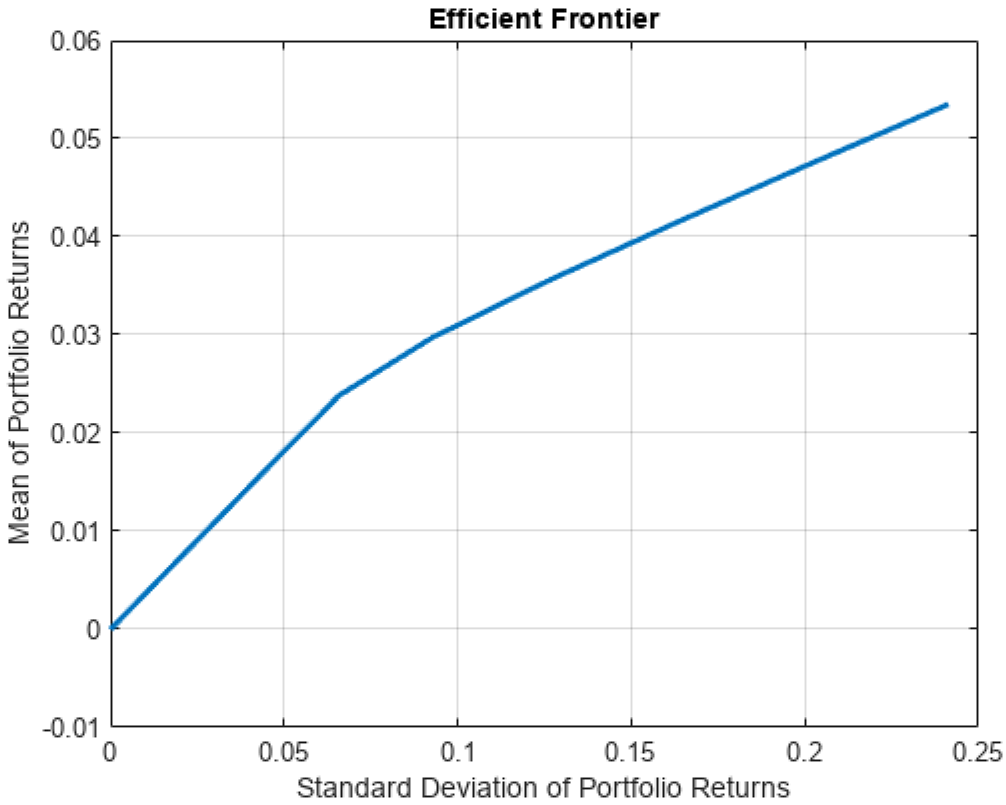
Use `abs2active` to compute the constraints for active portfolio weights.

```
ActCons = abs2active(AbsCons, w0)
```

```
ActCons = 8×4

    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 for a `Portfolio` object specifies weights to be nonnegative).

Input Arguments

`AbsConSet` — Portfolio linear inequality constraint matrix expressed in absolute weight format

matrix

Portfolio linear inequality constraint matrix expressed in absolute weight format, specified 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.

Data Types: `double`

`Index` — Index portfolio weights

vector

Index portfolio weights, specified as a `NASSETS`-by-`1` vector. The sum of the index weights must equal the total portfolio value (for example, a standard portfolio optimization imposes a sum-to-1 budget constraint).

Data Types: `double`

Output Arguments

`ActiveConSet` — Transformed portfolio linear inequality constraint matrix expressed in active weight format

matrix

Transformed portfolio linear inequality constraint matrix expressed in active weight format, returned in 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.

Algorithms

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

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

Therefore

$$Aw_{active} \leq b_{absolute} - Aw_{index} = b_{active}.$$

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

Version History

Introduced before R2006a

See Also

`active2abs` | `pcalims` | `pcglims` | `pcpval` | `portcons` | `Portfolio` | `setInequality`

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 = datetime(1997,3,14);
Maturity = [datetime(2000,11,30) datetime(2000,12,31) datetime(2001,1,31)];
Period = 2;
Basis = 0;
EndMonthRule = 1;

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

```
Fraction = 3×1

    0.5714
    0.4033
    0.2320
```

Input Arguments

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date, specified as a scalar, or an `NUMBONDS`-by-`1` vector using a datetime array, or string array, or date character vectors. `Settle` must be earlier than `Maturity`.

To support existing code, `accrfrac` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date, specified as a scalar, or an `NUMBONDS`-by-`1` vector using a datetime array, or string array, or date character vectors.

To support existing code, `accrfrac` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `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 scalar or an `NUMBONDS`-by-`1` 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 a scalar integer with a value of `0` through `13` or a `NUMBONDS`-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 2-16.

Data Types: single | double

EndMonthRule — End-of-month rule flag for month having 30 or fewer days

1 (in effect) (default) | nonnegative integer 0 or 1

End-of-month rule flag for month having 30 or fewer days, specified as a scalar nonnegative integer or an NUMBONDS-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

datetime array | string array | date character vector

Bond issue date, specified as a scalar, or an NUMBONDS-by-1 vector using a datetime array, or string array, or date character vectors.

To support existing code, accrfrac also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

FirstCouponDate — Date when bond makes first coupon payment

datetime array | string array | date character vector

Date when a bond makes its first coupon payment, specified as a scalar, or an NUMBONDS-by-1 vector using a datetime array, or string array, or date character vectors.

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.

To support existing code, accrfrac also accepts serial date numbers as inputs, but they are not recommended.

Data Types: double | char | string | datetime

LastCouponDate — Last coupon date of bond before maturity date

datetime array | string array | date character vector

Last coupon date of a bond before maturity date, specified as a scalar, or an NUMBONDS-by-1 vector using a datetime array, or string array, or date character vectors.

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.

To support existing code, accrfrac also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | 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.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although accrfrac supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

cfdates | cfamounts | cpncount | cpndaten | cpndatenq | cpndatep | cpndaysn | cpndaysp | cpnpersz | datetime | cpndatepq

Topics

"Pricing and Computing Yields for Fixed-Income Securities" on page 2-15

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(datetime(1983,1,31), datetime(1993,3,1),datetime(1983,7,31),  100,  0.1
```

```
AccruInterest = 0.8011
```

Input Arguments

`IssueDate` — Issue date of security
datetime array | string array | date character vector

Issue date of the security, specified as a scalar or a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `acrubond` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Settle` — Settlement date of security
datetime array | string array | date character vector

Settlement date of the security, specified as a scalar or a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors. The `Settle` date must be before the `Maturity` date.

To support existing code, `acrubond` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`FirstCouponDate` — First coupon date of security
datetime array | string array | date character vector

First coupon date of the security, specified as a scalar or a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `acrubond` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `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 2-16.

Data Types: double

Output Arguments

AccruInterest — Accrued interest

numeric

Accrued interest for the security, returned as a scalar or a NINST-by-1 vector.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although acrubond supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

accrfrac | acrudisc | bndprice | bndyield | cfamounts | datenum | datetime

Topics

“Coupon Date Calculations” on page 2-20
“Fixed-Income Terminology” on page 2-15

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(datetime(1992,5,1), datetime(1992,7,15), 100,  0.1,  2,  0)
```

```
AccruInterest = 2.0604
```

Input Arguments

`Settle` — Settlement date of security
datetime array | string array | date character vector

Settlement date of the security, specified as a scalar or a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors. The `Settle` date must be before the `Maturity` date.

To support existing code, `acrudisc` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Maturity` — Maturity date of security
datetime array | string array | date character vector

Maturity date of the security, specified as a scalar or a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `acrudisc` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `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 2-16.

Data Types: `double`

Output Arguments

`AccruInterest` — Accrued interest

numeric

Accrued interest, returned as a scalar or a `NINST`-by-`1` vector.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although `acrudisc` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

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

See Also

`acrubond` | `prdisc` | `prmat` | `ylddisc` | `yldmat` | `datetime`

Topics

"Coupon Date Calculations" on page 2-20
"Fixed-Income Terminology" on page 2-15

active2abs

Convert constraints from active to absolute format

Syntax

```
AbsConSet = active2abs(ActiveConSet,Index)
```

Description

`AbsConSet = active2abs(ActiveConSet,Index)` transforms a constraint matrix to an equivalent matrix expressed in absolute weight format. The transformation equation is

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

Therefore

$$Aw_{absolute} \leq b_{active} + Aw_{index} = b_{absolute} .$$

The initial constraint matrix consists of `NCONSTRAINTS` portfolio linear inequality constraints expressed in active weight format (relative to the index portfolio). The index portfolio vector contains `NASSETS` assets.

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

Input Arguments

`ActiveConSet` — Portfolio linear inequality constraint matrix expressed in active weight format

matrix

Portfolio linear inequality constraint matrix expressed in active weight format, 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.

Data Types: `double`

`Index` — Index of portfolio weights

vector

Index of portfolio weights, specified as an `ASSETS`-by-`1` vector. The sum of the index weights must equal the total portfolio value. For example, a standard portfolio optimization imposes a sum to `1` budget constraint.

Data Types: `double`

Output Arguments

`AbsConSet` — Transformed portfolio linear inequality constraint

matrix

Transformed portfolio linear inequality constraint, returned as a matrix and 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.

Version History

Introduced before R2006a

See Also

`abs2active` | `pcalims` | `pcgcomp` | `pcglims` | `pcpval` | `portcons`

addEquality

Add linear equality constraints for portfolio weights to existing constraints

Syntax

```
obj = addEquality(obj,AEquality,bEquality)
```

Description

`obj = addEquality(obj,AEquality,bEquality)` adds linear equality constraints for portfolio weights to existing constraints for `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` objects. For details on the respective workflows when using these different objects, see “Portfolio Object Workflow” on page 4-17, “PortfolioCVaR Object Workflow” on page 5-16, and “PortfolioMAD Object Workflow” on page 6-15.

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
```

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`

Data Types: `object`

`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([ ], [ ])
  ```

Version History

Introduced in R2011a

See Also

`setEquality`

Topics

"Working with Linear Equality Constraints Using Portfolio Object" on page 4-72
"Working with Linear Equality Constraints Using PortfolioCVaR Object" on page 5-65
"Setting Linear Inequality Constraints Using the setInequality and addInequality Functions" on page 6-65
"Portfolio Optimization Examples Using Financial Toolbox™" on page 4-152
"Portfolio Set for Optimization Using Portfolio Objects" on page 4-8
"Portfolio Set for Optimization Using PortfolioCVaR Object" on page 5-8
"Portfolio Set for Optimization Using PortfolioMAD Object" on page 6-7

addGroupRatio

Add group ratio constraints for portfolio weights to existing group ratio constraints

Syntax

```
obj = addGroupRatio(obj,GroupA,GroupB,LowerRatio)
obj = addGroupRatio(obj,GroupA,GroupB,LowerRatio,UpperRatio)
```

Description

`obj = addGroupRatio(obj,GroupA,GroupB,LowerRatio)` adds group ratio constraints for portfolio weights to existing group ratio constraints for `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` objects. For details on the respective workflows when using these different objects, see "Portfolio Object Workflow" on page 4-17, "PortfolioCVaR Object Workflow" on page 5-16, and "PortfolioMAD Object Workflow" on page 6-15.

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
```

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`

Data Types: `object`

`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.

Version History

Introduced in R2011a

See Also

`setGroupRatio`

Topics

"Working with Group Ratio Constraints Using Portfolio Object" on page 4-69
"Working with Group Ratio Constraints Using PortfolioCVaR Object" on page 5-62
"Working with Group Ratio Constraints Using PortfolioMAD Object" on page 6-60
"Portfolio Optimization Examples Using Financial Toolbox™" on page 4-152
"Portfolio Set for Optimization Using Portfolio Objects" on page 4-8
"Portfolio Set for Optimization Using PortfolioCVaR Object" on page 5-8
"Portfolio Set for Optimization Using PortfolioMAD Object" on page 6-7

addGroups

Add group constraints for portfolio weights to existing group constraints

Syntax

```
obj = addGroups(obj,GroupMatrix,LowerGroup)
obj = addGroups(obj,GroupMatrix,LowerGroup,UpperGroup)
```

Description

`obj = addGroups(obj,GroupMatrix,LowerGroup)` adds group constraints for portfolio weights to existing group constraints for `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` objects. For details on the respective workflows when using these different objects, see “Portfolio Object Workflow” on page 4-17, “PortfolioCVaR Object Workflow” on page 5-16, and “PortfolioMAD Object Workflow” on page 6-15.

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
```

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`

Data Types: `object`

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.

Version History

Introduced in R2011a

See Also

`setGroups`

Topics

“Working with Group Constraints Using Portfolio Object” on page 4-66
“Working with Group Constraints Using PortfolioCVaR Object” on page 5-59
“Working with Group Constraints Using PortfolioMAD Object” on page 6-57
“Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
“Portfolio Set for Optimization Using Portfolio Objects” on page 4-8
“Portfolio Set for Optimization Using PortfolioCVaR Object” on page 5-8
“Portfolio Set for Optimization Using PortfolioMAD Object” on page 6-7

addInequality

Add linear inequality constraints for portfolio weights to existing constraints

Syntax

```
obj = addInequality(obj,AInequality,bInequality)
```

Description

`obj = addInequality(obj,AInequality,bInequality)` adds linear inequality constraints for portfolio weights to existing constraints for `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` objects. For details on the respective workflows when using these different objects, see “Portfolio Object Workflow” on page 4-17, “PortfolioCVaR Object Workflow” on page 5-16, and “PortfolioMAD Object Workflow” on page 6-15.

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
```

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`

Data Types: `object`

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([ ], [ ])
  ```

Version History

Introduced in R2011a

See Also

`setInequality`

Topics

"Working with Linear Inequality Constraints Using Portfolio Object" on page 4-75
"Working with Linear Inequality Constraints Using PortfolioCVaR Object" on page 5-67
"Working with Linear Inequality Constraints Using PortfolioMAD Object" on page 6-65
"Portfolio Optimization Examples Using Financial Toolbox™" on page 4-152
"Portfolio Set for Optimization Using Portfolio Objects" on page 4-8
"Portfolio Set for Optimization Using PortfolioCVaR Object" on page 5-8
"Portfolio Set for Optimization Using PortfolioMAD Object" on page 6-7

adline

Accumulation/Distribution line

Syntax

```
ADline = adline(Data)
```

Description

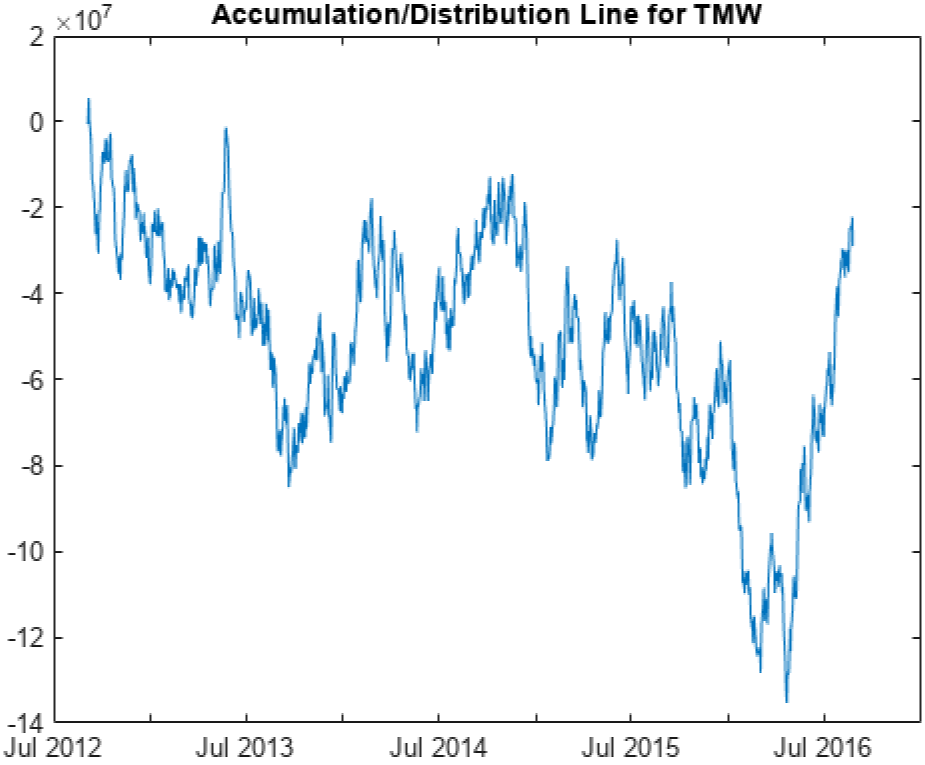
`ADline = adline(Data)` calculates the Accumulation/Distribution Line from a set of high, low, closing prices, and volume traded of a security.

Examples

Calculate the Accumulation/Distribution Line for a Data Series for a Stock

Load the file `SimulatedStock.mat`, which provides a timetable (`TMW`) for financial data for TMW stock.

```
load SimulatedStock.mat
line = adline(TMW);
plot(line.Time,line.ADLine)
title('Accumulation/Distribution Line for TMW')
```



Input Arguments

`Data` — Data with high, low, closing prices, and volume traded

matrix | table | timetable

Data with high, low, closing prices and volume traded, specified as a matrix, table, or timetable. For matrix input, `Data` is `M`-by-4 with high, low, closing prices, and volume traded. Timetables and tables with `M` rows must contain a variable named `'High'`, `'Low'`, `'Close'`, and `'Volume'` (case insensitive).

Data Types: `double` | `table` | `timetable`

Output Arguments

`ADline` — Accumulation/Distribution line

matrix | table | timetable

Accumulation/Distribution line, returned with the same number of rows (`M`) and the same type (matrix, table, or timetable) as the input `Data`.

Version History

Introduced before R2006a

R2023a: `fints` support removed for `Data` input argument

Behavior changed in R2023a

`fints` object support for the `Data` input argument is removed.

R2022b: Support for negative price data

Behavior changed in R2022b

The `Data` input accepts negative prices.

References

[1] Achelis, S. B. *Technical Analysis from A to Z.* Second Edition. McGraw-Hill, 1995, pp. 52–53.

See Also

`timetable` | `table` | `adosc` | `willad` | `willpctr`

Topics

"Use Timetables in Finance" on page 11-7
"Convert Financial Time Series Objects (fints) to Timetables" on page 11-2

adosc

Accumulation/Distribution oscillator

Syntax

```
ado = adosc(Data)
```

Description

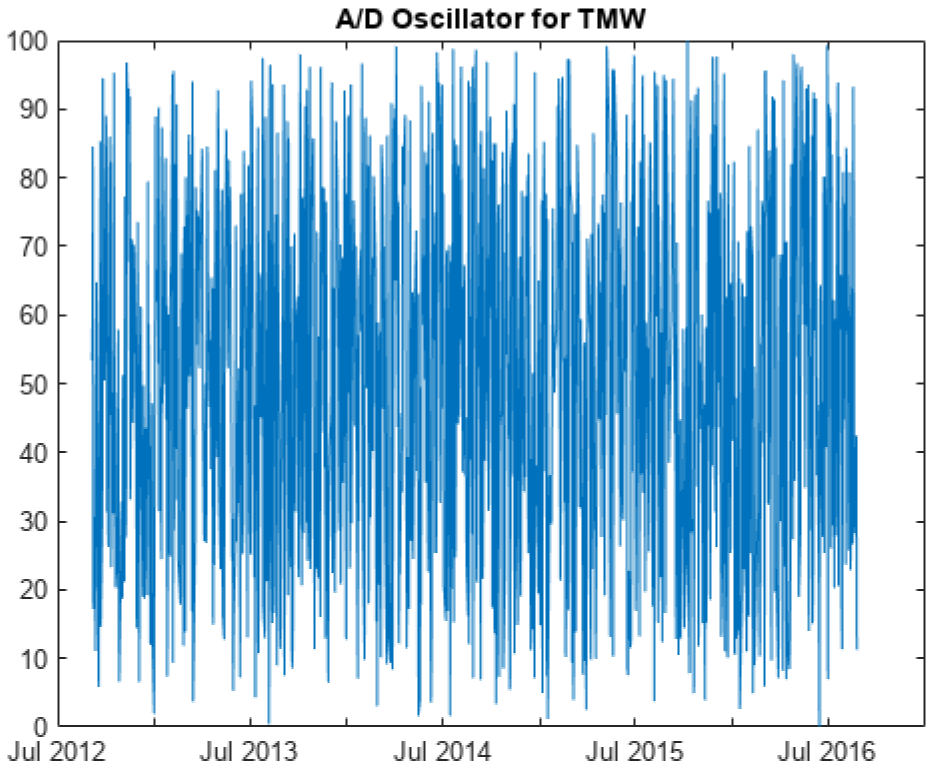
`ado = adosc(Data)` calculates the Accumulation/Distribution (A/D) oscillator.

Examples

Calculate the Accumulation/Distribution Oscillator for a Stock

Load the file `SimulatedStock.mat`, which provides a timetable (`TMW`) for financial data for TMW stock.

```
load SimulatedStock.mat
ADOsc = adosc(TMW);
plot(ADOsc.dates, ADOsc.ADOscillator)
title('A/D Oscillator for TMW')
```



Input Arguments

Data — Data with open, high, low, close information
matrix | table | timetable

Data with open, high, low, close information, specified as a matrix, table, or timetable. For matrix input, `Data` is an `M`-by-4 matrix of open, high, low, and closing prices. Timetables and tables with `M` rows must contain variables named `'Open'`, `'High'`, `'Low'`, and `'Close'` (case insensitive).

Data Types: `double` | `table` | `timetable`

Output Arguments

ado — Accumulation/Distribution oscillator
matrix | table | timetable

Accumulation/Distribution oscillator, returned with the same number of rows (`M`) and type (matrix, table, or timetable) as the input `Data`.

Version History

Introduced before R2006a

R2023a: `fints` support removed for `Data` input argument
Behavior changed in R2023a

`fints` object support for the `Data` input argument is removed.

R2022b: Support for negative price data
Behavior changed in R2022b

The `Data` input accepts negative prices.

References

[1] Kaufman, P. J. *The New Commodity Trading Systems and Methods*. John Wiley and Sons, New York, 1987.

See Also

`adline` | `willad` | `timetable` | `table`

Topics
“Use Timetables in Finance” on page 11-7
“Convert Financial Time Series Objects (fints) to Timetables” on page 11-2

amortize

Amortization schedule

Syntax

```
[Principal,Interest,Balance,Payment] = amortize(Rate,NumPeriods,PresentValue)
[Principal,Interest,Balance,Payment] = amortize( ___ ,FutureValue,Due)
```

Description

`[Principal,Interest,Balance,Payment] = amortize(Rate,NumPeriods,PresentValue)` returns the principal and interest payments of a loan, the remaining balance of the original loan amount, and the periodic payment.

`[Principal,Interest,Balance,Payment] = amortize( ___ ,FutureValue,Due)` specifies options using one or more optional arguments in addition to the input arguments in the previous syntax.

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);
```

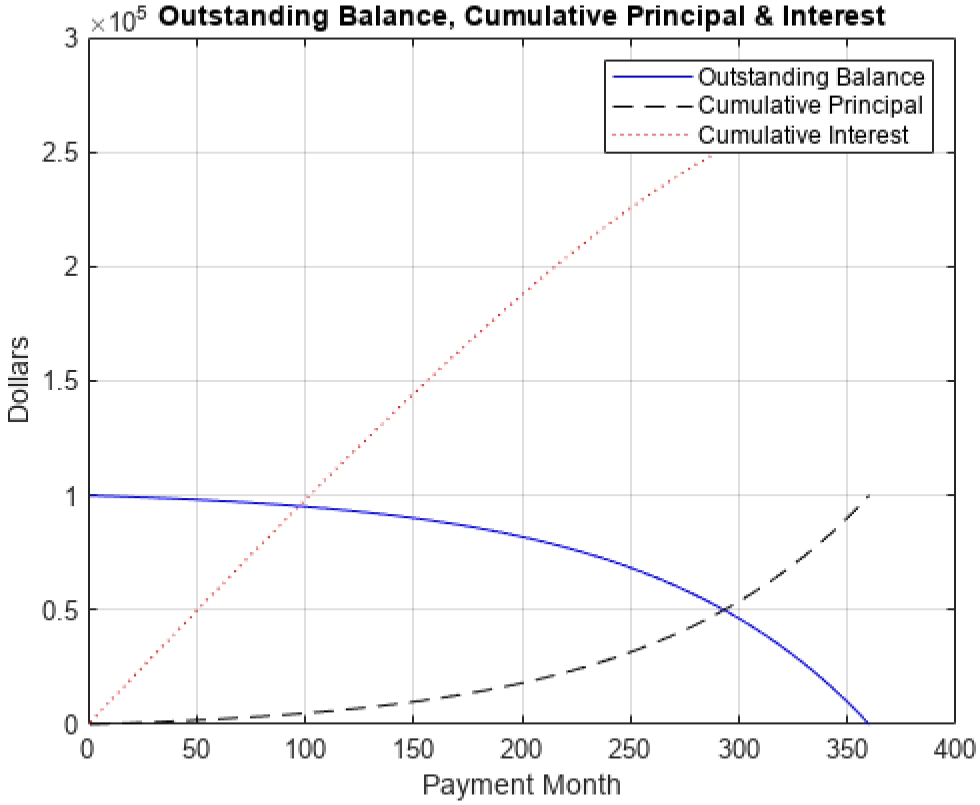
The output argument `Payment` contains the fixed monthly payment.

```
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.

Input Arguments

Rate — Interest-rate per period

scalar numeric decimal

Interest-rate per period, specified as a scalar numeric decimal.

Data Types: `double`

`NumPeriods` — Number of payment periods

scalar numeric

Number of payment periods, specified as a scalar numeric.

Data Types: `double`

`PresentValue` — Present value of the loan

scalar numeric

Present value of the loan, specified as a scalar numeric.

Data Types: `double`

`FutureValue` — Future value of the loan

`0` (default) | scalar numeric

(Optional) Future value of the loan, specified as a scalar numeric.

Data Types: `double`

`Due` — When payments are due

`0` (end of period) (default) | scalar integer with value of `0` or `1`

(Optional) When payments are due, specified as a scalar integer with value of `0` (end of period) or `1` (beginning of period).

Data Types: `double`

Output Arguments

`Principal` — Principal paid in each period

vector

Principal paid in each period, returned as a `1`-by-`NumPeriods` vector.

`Interest` — Interest paid in each period

vector

Interest paid in each period, returned as a `1`-by-`NumPeriods` vector.

`Balance` — Remaining balance of the loan in each payment period

vector

Remaining balance of the loan in each payment period, returned as a `1`-by-`NumPeriods` vector.

`Payment` — Payment per period

scalar numeric

Payment per period, returned as a scalar numeric.

Version History

Introduced before R2006a

See Also

`annurate` | `annuterm` | `payadv` | `payodd` | `payper`

Topics

"Analyzing and Computing Cash Flows" on page 2-11

annurate

Periodic interest rate of annuity

Syntax

```
Rate = annurate(NumPeriods,Payment,PresentValue)
Rate = annurate( ___ ,FutureValue,Due)
```

Description

`Rate = annurate(NumPeriods,Payment,PresentValue)` returns the periodic interest rate paid on a loan or annuity.

`Rate = annurate( ___ ,FutureValue,Due)` specifies options using one or more optional arguments in addition to the input arguments in the previous syntax.

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.

Input Arguments

`NumPeriods` — Number of payment periods
scalar numeric

Number of payment periods, specified as a scalar numeric.

Data Types: `double`

`Payment` — Payment per period
scalar numeric

Payment per period, specified as a scalar numeric.

Data Types: `double`

`PresentValue` — Present value of the loan
scalar numeric

Present value of the loan, specified as a scalar numeric.

Data Types: `double`

`FutureValue` — Future value of the loan

`0` (default) | scalar numeric

(Optional) Future value of the loan, specified as a scalar numeric.

Data Types: `double`

`Due` — When payments are due

`0` (end of period) (default) | scalar integer with value of `0` or `1`

(Optional) When payments are due, specified as a scalar integer with value of `0` (end of period) or `1` (beginning of period).

Data Types: `double`

Output Arguments

`Rate` — Periodic interest-rate paid of a loan or annuity

scalar numeric decimal

Periodic interest-rate paid of a loan or annuity, returned as a scalar numeric decimal.

Version History

Introduced before R2006a

See Also

`amortize` | `annuterm` | `bndyield` | `irr`

Topics

"Analyzing and Computing Cash Flows" on page 2-11

annuterm

Number of periods to obtain value

Syntax

```
NumPeriods = annuterm(Rate,Payment,PresentValue)
NumPeriods = annuterm( ___ ,FutureValue,Due)
```

Description

`NumPeriods = annuterm(Rate,Payment,PresentValue)` 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.

`NumPeriods = annuterm( ___ ,FutureValue,Due)` specifies options using one or more optional arguments in addition to the input arguments in the previous syntax.

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 months will it take to save $5,000?

```
NumPeriods = annuterm(0.09/12, 200, 1500, 5000, 0)
```

```
NumPeriods = 15.6752
```

Input Arguments

`Rate` — Interest-rate per period
scalar numeric decimal

Interest-rate per period, specified as a scalar numeric decimal.

Data Types: `double`

`Payment` — Payment per period
scalar numeric

Payment per period, specified as a scalar numeric.

Data Types: `double`

`PresentValue` — Present value of the loan
scalar numeric

Present value of the loan, specified as a scalar numeric.

Data Types: `double`

`FutureValue` — Future value of the loan
`0` (default) | scalar numeric

(Optional) Future value of the loan, specified as a scalar numeric.

Data Types: `double`

`Due` — When payments are due
`0` (end of period) (default) | scalar integer with value of `0` or `1`

(Optional) When payments are due, specified as a scalar integer with value of `0` (end of period) or `1` (beginning of period).

Data Types: `double`

Output Arguments

`NumPeriods` — Number of payment periods
scalar numeric

Number of payment periods, returned as a scalar numeric.

Version History

Introduced before R2006a

See Also

`annurate` | `amortize` | `fvfix` | `pvfix`

Topics

"Analyzing and Computing Cash Flows" on page 2-11

arith2geom

Arithmetic to geometric moments of asset returns

Syntax

```
[mg,Cg = arith2geom(ma,Ca)
[mg,Cg = arith2geom( ___ ,t)
```

Description

`[mg,Cg = arith2geom(ma,Ca)` 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( ___ ,t)` adds an optional argument `t`.

Examples

Obtain Arithmetic to Geometric Moments of Asset Returns

This example shows several variations of using `arith2geom`.

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 the optional input `t = 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 ];
[mg, Cg] = arith2geom(m, C, 12)
```

```
mg = 4×1

    0.8934
    2.9488
    4.9632
   17.0835


Cg = 4×4
10^3 ×

    0.0003    0.0004    0.0003         0
    0.0004    0.0065    0.0065    0.0110
    0.0003    0.0065    0.0354    0.0536
         0    0.0110    0.0536    1.0952
```

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 the optional input `t = 1/12`.

```
[mg, Cg] = arith2geom(m, C, 1/12)
```

```
mg = 4×1

    0.0044
    0.0096
    0.0125
    0.0203


Cg = 4×4

    0.0005    0.0003    0.0002         0
    0.0003    0.0025    0.0017    0.0010
    0.0002    0.0017    0.0049    0.0029
         0    0.0010    0.0029    0.0107
```

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 the optional input `t = 3`.

```
[mg, Cg] = arith2geom(m, C, 3)
```

```
mg = 4×1

    0.1730
    0.4097
    0.5627
    1.0622


Cg = 4×4

    0.0267    0.0204    0.0106         0
    0.0204    0.1800    0.1390    0.1057
    0.0106    0.1390    0.4606    0.3418
         0    0.1057    0.3418    1.8886
```

Input Arguments

`ma` — Arithmetic mean of asset-return data

vector

Arithmetic mean of asset-return data, specified as an *n*-vector.

Data Types: `double`

`Ca` — Arithmetic covariance of asset-return data

matrix

Arithmetic covariance of asset-return data, specified as an n-by-n symmetric, positive semidefinite matrix. If Ca is not a symmetric positive semidefinite matrix, use nearcorr to create a positive semidefinite matrix for a correlation matrix.

Data Types: double

t — Target period of geometric moments in terms of periodicity of arithmetic moments
1 (default) | scalar positive numeric

(Optional) Target period of geometric moments in terms of periodicity of arithmetic moments, specified as a scalar positive numeric.

Data Types: double

Output Arguments

mg — Continuously compounded or "geometric" mean of asset returns over the target period
vector

Continuously compounded or "geometric" mean of asset returns over the target period (t), returned as an *n*-vector.

Cg — Continuously compounded or "geometric" covariance of asset returns over the target period
matrix

Continuously compounded or "geometric" covariance of asset returns over the target period (t), returned as an n-by-n matrix.

Algorithms

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

$$1 + \mathrm{m}_{G_i} = \exp(t\mathrm{m}_{A_i} + \frac{1}{2}t\mathrm{C}_{A_{ii}})$$

$$\mathrm{C}_{G_{ij}} = (1 + \mathrm{m}_{G_i})(1 + \mathrm{m}_{G_\mathrm{j}})(\exp(t\mathrm{C}_{Aij}) - 1)$$

For $i,j = 1,\ldots, n$.

Note If $t = 1$, then $\mathbf{Y} = \exp(\mathbf{X})$.

The `arith2geom` 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`.

Version History

Introduced before R2006a

See Also

`geom2arith` | `nearcorr`

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(datetime(2000,2,11),datetime(2000,8,7), 0.0577)
```

```
Yield = 0.0602
```

Input Arguments

`Settle` — Settlement date of Treasury bill
datetime array | string array | date character vector

Settlement date of the Treasury bill, specified as a scalar or a `NTBILLS`-by-`1` vector using a datetime array, string array, or date character vectors. `Settle` must be earlier than `Maturity`.

To support existing code, `beytbill` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Maturity` — Maturity date of Treasury bill
datetime array | string array | date character vector

Maturity date of the Treasury bill, specified as a scalar or a `NTBILLS`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `beytbill` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `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`.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although `beytbill` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`datenum` | `prtbill` | `yldtbill` | `datetime`

Topics

"Computing Treasury Bill Price and Yield" on page 2-26
"Treasury Bills Defined" on page 2-25

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. An American option can be exercised any time until its expiration date.

`[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 = 6×6

   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 = 6×6

    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.

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.

Version History

Introduced before R2006a

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 2-39
“Greek-Neutral Portfolios of European Stock Options” on page 10-14
“Plotting Sensitivities of an Option” on page 10-25
“Plotting Sensitivities of a Portfolio of Options” on page 10-27

blkimpv

Implied volatility for futures options from Black model

Syntax

```
Volatility = blkimpv(Price,Strike,Rate,Time,Value)
Volatility = blkimpv( ___ ,Name,Value)
```

Description

`Volatility = blkimpv(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. If the `Class` name-value argument is empty or unspecified, the default is a call option

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( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

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, 'Limit',0.5);
Volatility = blkimpv(20, 20, 0.09, 4/12, 1.1166, 'Limit',0.5,'Class',{'Call'});
Volatility = blkimpv(20, 20, 0.09, 4/12, 1.1166, 'Limit',0.5,'Class',true);
Volatility = blkimpv(20, 20, 0.09, 4/12, 1.1166, 'Limit',0.5,'Class',true,'Method','jackel2016')
```

```
Volatility = 0.2500
```

Input Arguments

`Price` — Current price of underlying asset

scalar numeric

Current price of the underlying asset (that is, a futures contract), specified as a scalar numeric.

Data Types: `double`

`Strike` — Exercise price of the futures option

scalar numeric

Exercise price of the futures option, specified as a scalar numeric.

Data Types: `double`

`Rate` — Annualized continuously compounded risk-free rate of return over life of the option

scalar positive decimal

Annualized continuously compounded risk-free rate of return over the life of the option, specified as a scalar positive decimal.

Data Types: `double`

`Time` — Time to expiration of the futures option

scalar numeric

Time to expiration of the futures option, specified as the number of years using a scalar numeric.

Data Types: `double`

`Value` — Price of a European option from which implied volatility of underlying asset is derived

scalar numeric

Price of a European futures option from which the implied volatility of the underlying asset is derived, specified as a scalar numeric.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Volatility = blkimpv(Yield,CouponRate,Settle,Maturity,'Method','jackel2016')`

`Limit` — Upper bound of implied volatility search interval

`10` (1000% per annum) (default) | positive scalar numeric

Upper bound of the implied volatility search interval, specified as the comma-separated pair consisting of `'Limit'` and a positive scalar numeric. If `Limit` is empty or unspecified, the default is `10`, or 1000% per annum.

Note If you are using `Method` with a value of `'jackel2016'`, the `Limit` argument is ignored.

Data Types: `double`

`Tolerance` — Implied volatility termination tolerance

`1e6` (default) | positive scalar numeric

Implied volatility termination tolerance, specified as the comma-separated pair consisting of `'Tolerance'` and a positive scalar numeric. If empty or missing, the default is `1e6`.

Note If you are using `Method` with a value of `'jackel2016'`, the `Tolerance` argument is ignored.

Data Types: `double`

`Class` — Option class from which implied volatility is derived

`true` (call option) (default) | logical | cell array of character vectors | string array

Option class indicating option type (call or put) from which implied volatility is derived, specified as the comma-separated pair consisting of `'Class'` and a logical indicator, cell array of character vectors, or string array.

To specify call options, set `Class = true` or `Class = {'call'}`. To specify put options, set `Class = false` or `Class = {'put'}` or `Class = ["put"]`. If `Class` is empty or unspecified, the default is a call option.

Data Types: `logical` | `cell` | `string`

`Method` — Method for computing implied volatility

`'jackel2016'` (default) | character vector with values `'search'` or `'jackel2016'` | string with values `"search"` or `"jackel2016"`

Method for computing implied volatility, specified as the comma-separated pair consisting of `'Method'` and a character vector with a value of `'search'` or `'jackel2016'` or a string with a value of `"search"` or `"jackel2016"`.

Data Types: `char` | `string`

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. If no solution is found, `blkimpv` returns `NaN`.

Version History

Introduced before R2006a

References

[1] Hull, John C. *Options, Futures, and Other Derivatives. 5th edition*, Prentice Hall, 2003, pp. 287–288.

[2] Jäckel, Peter. "Let's Be Rational." *Wilmott Magazine.*, January, 2015 (https://onlinelibrary.wiley.com/doi/abs/10.1002/wilm.10395).

[3] Black, Fischer. “The Pricing of Commodity Contracts.” *Journal of Financial Economics.* March 3, 1976, pp. 167–79.

See Also

`blkprice` | `blsprice` | `blsimpv`

Topics

“Pricing and Analyzing Equity Derivatives” on page 2-39
“Greek-Neutral Portfolios of European Stock Options” on page 10-14
“Plotting Sensitivities of an Option” on page 10-25
“Plotting Sensitivities of a Portfolio of Options” on page 10-27

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
```

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.

Version History

Introduced before R2006a

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 2-39
"Greek-Neutral Portfolios of European Stock Options" on page 10-14
"Plotting Sensitivities of an Option" on page 10-25

"Plotting Sensitivities of a Portfolio of Options" on page 10-27

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.

In addition, you can use the Financial Instruments Toolbox object framework with the `BlackScholes` pricer object to obtain price and `delta` values for a `Vanilla`, `Barrier`, `Touch`, `DoubleTouch`, or `Binary` instrument using a `BlackScholes` model.

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
```

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 value.

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.

Version History

Introduced in R2006a

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 2-39
"Greek-Neutral Portfolios of European Stock Options" on page 10-14
"Plotting Sensitivities of an Option" on page 10-25
"Plotting Sensitivities of a Portfolio of Options" on page 10-27

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.

In addition, you can use the Financial Instruments Toolbox object framework with the `BlackScholes` pricer object to obtain price and `gamma` values for a `Vanilla`, `Barrier`, `Touch`, `DoubleTouch`, or `Binary` instrument using a `BlackScholes` model.

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
```

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 value.

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.

Version History
Introduced in R2006a

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 2-39
“Greek-Neutral Portfolios of European Stock Options” on page 10-14
“Plotting Sensitivities of an Option” on page 10-25
“Plotting Sensitivities of a Portfolio of Options” on page 10-27

blsimpv

Black-Scholes implied volatility

Syntax

```
Volatility = blsimpv(Price,Strike,Rate,Time,Value)
Volatility = blsimpv( ___ ,Name,Value)
```

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 options. If the `Class` name-value argument is empty or unspecified, the default is a call option

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( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

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, 'Limit', 0.5);
Volatility = blsimpv(100, 95, 0.075, 0.25, 10, 'Limit',0.5,'Yield',0,'Class', {'Call'});
Volatility = blsimpv(100, 95, 0.075, 0.25, 10, 'Limit',0.5,'Yield',0, 'Class', true);
Volatility = blsimpv(100, 95, 0.075, 0.25, 10, 'Limit',0.5,'Yield',0, 'Class', true,'Method','ja
```

```
Volatility = 0.3130
```

Input Arguments

`Price` — Current price of underlying asset

scalar numeric

Current price of the underlying asset, specified as a scalar numeric.

Data Types: `double`

`Strike` — Exercise price of the option
scalar numeric

Exercise price of the option, specified as a scalar numeric.

Data Types: `double`

`Rate` — Annualized continuously compounded risk-free rate of return over life of the option
scalar positive decimal

Annualized continuously compounded risk-free rate of return over the life of the option, specified as a scalar positive decimal.

Data Types: `double`

`Time` — Time to expiration of option
scalar numeric

Time to expiration of the option, specified as the number of years using a scalar numeric.

Data Types: `double`

`Value` — Price of a European option from which implied volatility of underlying asset is derived
scalar numeric

Price of a European option from which the implied volatility of the underlying asset is derived, specified as a scalar numeric.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Volatility = blsimpv(Yield,CouponRate,Settle,Maturity,'Method','jackel2016')`

`Limit` — Upper bound of implied volatility search interval
`10` (1000% per annum) (default) | positive scalar numeric

Upper bound of the implied volatility search interval, specified as the comma-separated pair consisting of `'Limit'` and a positive scalar numeric. If `Limit` is empty or unspecified, the default is `10`, or 1000% per annum.

Note If you are using `Method` with a value of `'jackel2016'`, the `Limit` argument is ignored.

Data Types: `double`

`Yield` — Annualized continuously compounded yield of underlying asset over life of the option

`0` (default) | decimal

Annualized continuously compounded yield of the underlying asset over the life of the option,specified as the comma-separated pair consisting of `'Yield'` and 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

`1e6` (default) | positive scalar numeric

Implied volatility termination tolerance, specified as the comma-separated pair consisting of `'Tolerance'` and a positive scalar numeric. If empty or missing, the default is `1e6`.

Note If you are using `Method` with a value of `'jackel2016'`, the `Tolerance` argument is ignored.

Data Types: `double`

`Class` — Option class from which implied volatility is derived

`true` (call option) (default) | logical | cell array of character vectors | string array

Option class indicating option type (call or put) from which implied volatility is derived, specified as the comma-separated pair consisting of `'Class'` and a logical indicator, cell array of character vectors, or string array.

To specify call options, set `Class = true` or `Class = {'call'}`. To specify put options, set `Class = false` or `Class = {'put'}` or `Class = ["put"]`. If `Class` is empty or unspecified, the default is a call option.

Data Types: `logical` | `cell` | `string`

`Method` — Method for computing implied volatility

`'jackel2016'` (default) | character vector with values `'search'` or `'jackel2016'` | string with values `"search"` or `"jackel2016"`

Method for computing implied volatility, specified as the comma-separated pair consisting of `'Method'` and a character vector with a value of `'search'` or `'jackel2016'` or a string with a value of `"search"` or `"jackel2016"`.

Data Types: `char` | `string`

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. If no solution is found, `blsimpv` returns `NaN`.

Version History

Introduced before R2006a

References

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

[2] Jäckel, Peter. "Let's Be Rational." *Wilmott Magazine.*, January, 2015 (https://onlinelibrary.wiley.com/doi/abs/10.1002/wilm.10395).

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

See Also

`blsgamma` | `blsdelta` | `blslambda` | `blsprice` | `blsrho` | `blstheta` | `blsvega`

Topics

“Pricing and Analyzing Equity Derivatives” on page 2-39
“Greek-Neutral Portfolios of European Stock Options” on page 10-14
“Plotting Sensitivities of an Option” on page 10-25
“Plotting Sensitivities of a Portfolio of Options” on page 10-27

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.

In addition, you can use the Financial Instruments Toolbox object framework with the `BlackScholes` pricer object to obtain price and `lambda` values for a `Vanilla`, `Barrier`, `Touch`, `DoubleTouch`, or `Binary` instrument using a `BlackScholes` model.

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
```

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 value.

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.

Version History

Introduced in R2006a

References

[1] Daigler, R. *Advanced Options Trading*. McGraw-Hill, 1993.

See Also

`blsgamma` | `blsdelta` | `blsprice` | `blsrho` | `blstheta` | `blsvega`

Topics

“Pricing and Analyzing Equity Derivatives” on page 2-39
“Greek-Neutral Portfolios of European Stock Options” on page 10-14
“Plotting Sensitivities of an Option” on page 10-25
“Plotting Sensitivities of a Portfolio of Options” on page 10-27

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.

In addition, you can use the Financial Instruments Toolbox object framework with the `BlackScholes` pricer object to obtain price values for a `Vanilla`, `Barrier`, `Touch`, `DoubleTouch`, or `Binary` instrument using a `BlackScholes` model.

`[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
```

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.

Version History

Introduced in R2006a

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

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

Topics

“Pricing and Analyzing Equity Derivatives” on page 2-39
“Greek-Neutral Portfolios of European Stock Options” on page 10-14
“Plotting Sensitivities of an Option” on page 10-25
“Plotting Sensitivities of a Portfolio of Options” on page 10-27

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.

In addition, you can use the Financial Instruments Toolbox object framework with the `BlackScholes` pricer object to obtain price and `rho` values for a `Vanilla`, `Barrier`, `Touch`, `DoubleTouch`, or `Binary` instrument using a `BlackScholes` model.

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
```

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 value.

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.

Version History

Introduced in R2006a

References

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

See Also

`blsgamma` | `blsdelta` | `blslambda` | `blsprice` | `blstheta` | `blsvega`

Topics

"Pricing and Analyzing Equity Derivatives" on page 2-39
"Greek-Neutral Portfolios of European Stock Options" on page 10-14
"Plotting Sensitivities of an Option" on page 10-25
"Plotting Sensitivities of a Portfolio of Options" on page 10-27

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 density function, in the Statistics and Machine Learning Toolbox.

In addition, you can use the Financial Instruments Toolbox object framework with the `BlackScholes` pricer object to obtain price and `theta` values for a `Vanilla`, `Barrier`, `Touch`, `DoubleTouch`, or `Binary` instrument using a `BlackScholes` model.

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
```

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 value.

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.

Version History

Introduced in R2006a

References

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

See Also

`blslambda` | `blsgamma` | `blsdelta` | `blsprice` | `blsrho` | `blsvega`

Topics

"Pricing and Analyzing Equity Derivatives" on page 2-39
"Greek-Neutral Portfolios of European Stock Options" on page 10-14
"Plotting Sensitivities of an Option" on page 10-25
"Plotting Sensitivities of a Portfolio of Options" on page 10-27

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)` returns the rate of change of the option value with respect to the volatility of the underlying asset. `blsvega` uses `normpdf`, the normal probability density function in the Statistics and Machine Learning Toolbox.

In addition, you can use the Financial Instruments Toolbox object framework with the `BlackScholes` pricer object to obtain price and `vega` values for a `Vanilla`, `Barrier`, `Touch`, `DoubleTouch`, or `Binary` instrument using a `BlackScholes` model.

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
```

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 value.

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.

Version History

Introduced in R2006a

References

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

See Also

`blslambda` | `blsgamma` | `blsdelta` | `blsprice` | `blsrho` | `blstheta`

Topics

“Pricing and Analyzing Equity Derivatives” on page 2-39
“Greek-Neutral Portfolios of European Stock Options” on page 10-14
“Plotting Sensitivities of an Option” on page 10-25
“Plotting Sensitivities of a Portfolio of Options” on page 10-27

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', '08/04/2025'}','mm
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)
```

```
ProbabilityData = 5×2

                    737226        0.0299675399937611
                    737433        0.0418832295824674
                    738188         0.090518332884262
                    738233         0.101248065083713
                    739833         0.233002708031915


HazardData = 5×2

                    737226        0.0157077745460244
                    737433        0.0217939816590403
                    738188         0.025184912824721
                    738233        0.0962608718640789
                    739833        0.0361632398787917
```

In bondDefaultBootstrap, the first column of the ProbabilityData output and the first column of the HazardData output contain the respective ending dates for the corresponding default probabilities and hazard rates. However, the starting dates used for the computation of the time ranges for default probabilities can be different from those of hazard rates. For default probabilities, the time ranges are all computed from the Settle date to the respective end dates shown in the first column of ProbabilityData. In contrast, the time ranges for the hazard rates are computed using the Settle date and the first column of HazardData, so that the first hazard rate applies from the Settle date to the first market date, the second hazard rate from the first to the second market date, and so on, and the last hazard rate applies from the second-to-last market date onwards.

```
datestr(Settle)

ans = 
'08-Jul-2016'

datestr(ProbabilityData(:,1))

ans = 5x11 char array
    '15-Jun-2018'
    '08-Jan-2019'
    '01-Feb-2021'
    '18-Mar-2021'
    '04-Aug-2025'


datestr(HazardData(:,1))

ans = 5x11 char array
    '15-Jun-2018'
    '08-Jan-2019'
    '01-Feb-2021'
    '18-Mar-2021'
    '04-Aug-2025'
```

The time ranges for the default probabilities all start on '08-Jul-2016' and they end on '15-Jun-2018', '08-Jan-2019', '01-Feb-2021', '18-Mar-2021', and '04-Aug-2025', respectively. As for the hazard rates, the first hazard rate starts on '08-Jul-2016' and ends on '15-Jun-2018', the second hazard rate starts on '15-Jun-2018' and ends on '08-Jan-2019', the third hazard rate starts on '08-Jan-2019' and ends on '01-Feb-2021', and so forth.

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=4×3 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=11×2 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)
```

```
defaultProb1 = 4×2

                    736847        0.0704863142317494
                    737577         0.162569420050034
                    737943         0.217308133826188
                    738673          0.38956773145021


hazard1 = 4×2

                    736847        0.0813390794774647
                    737577        0.0521615800986281
                    737943        0.0674145844133183
                    738673          0.12428587278862
```

```
format
```

Reformat the default probability and hazard rate for a better representation.

```
DefProbHazard = [defaultProb1, hazard1(:,2)];
DefProbHazardTable = array2table(DefProbHazard, 'VariableNames', {'Date', 'DefaultProbability', 
DefProbHazardTable.Date = datetime(DefProbHazardTable.Date,'Locale','en_US','ConvertFrom','datenu
DefProbHazardTable.Date.Format = 'MMM-dd-yyyy'
```

```
DefProbHazardTable=4×3 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=1×3 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)
```

```
PaymentDates = 1x6 datetime
   01-Dec-2016   01-Jun-2017   01-Dec-2017   01-Jun-2018   01-Dec-2018   01-Jun-2019
```

```
Payments = Payments(2:end)
```

```
Payments = 1×6

     4     4     4     4     4   104
```

Calculate the discount factors on the payment dates.

```
DF = zero2disc(interp1(RiskFreeRate.Date, RiskFreeRate.Rate, PaymentDates, 'linear', 'extrap'), P
```

```
DF = 1×6

    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 = 1×6

    40    40    40    40    40    40
```

Calculate the probability of default based on the default curve.

```
DefaultProb1 = bondDefaultBootstrap(ZeroData, MarketData, Settle, 'ZeroCompounding', -1, 'Probab
SurvivalProb = 1 - DefaultProb1(:,2)
```

```
SurvivalProb = 6×1

    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;SurvivalProb]));
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=1×5 table
     Maturity      Price     Coupon    ModelPrice    Difference
    ___________    ______    ______    __________    __________

    Jun-01-2019    109.02     0.08       109.02      1.4211e-14
```

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

datetime scalar | string scalar | date character vector

Settlement date, specified as a scalar datetime, string, or date character vector. Settle must be earlier than or equal to the maturity dates in MarketData.

To support existing code, bondDefaultBootstrap also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | datetime | string

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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) | datetime array | string array | date character vector

Dates for the output of default probability data, specified as the comma-separated pair consisting of `'ProbabilityDates'` and a `P`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `bondDefaultBootstrap` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `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 2-16.

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) | datetime array | string array | date character vector

Bond issue date, specified as the comma-separated pair consisting of 'IssueDate' and a N-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, bondDefaultBootstrap also accepts serial date numbers as inputs, but they are not recommended.

Data Types: 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) | datetime scalar | string scalar | date character vector

First actual coupon date, specified as the comma-separated pair consisting of 'FirstCouponDate' and a scalar datetime, string, or date character vector. 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.

To support existing code, bondDefaultBootstrap also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | datetime | string

LastCouponDate — Last actual coupon date

if you do not specify a LastCouponDate, cash flow payment dates are determined from other inputs (default) | datetime scalar | string scalar | date character vector

Last actual coupon date, specified as the comma-separated pair consisting of `'LastCouponDate'` and a scalar datetime, string, or date character vector. `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.

To support existing code, `bondDefaultBootstrap` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `datetime` | `string`

`StartDate` — Forward starting date of payments
if you do not specify `StartDate`, effective start date is `Settle` date (default) | datetime scalar | string scalar | date character vector

Forward starting date of payments, specified as the comma-separated pair consisting of `'StartDate'` and a scalar datetime, string, or date character vector. `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.

To support existing code, `bondDefaultBootstrap` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `datetime` | `string`

`BusinessDayConvention` — Business day conventions
`'actual'` (default) | character vector or string object 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.

More About

Bootstrap Default Probability

A default probability curve can be bootstrapped from a collection of bond market quotes.

Extracting discrete default probabilities for a certain period from market bond data is represented by the formula

$$Price = Disc(t_N) \times FV \times Q(t_N) + \frac{C}{f} \times \sum_{i=1}^{N} Disc(t_i) \times Q(t_i) + \sum_{(i=1)}^{N} Disc(t_i) \times R(t_i) \times (Q(t_i - 1) - Q(t_i))$$

where:

FV — Face value

Q — Survival probability

C — Coupon

R — Recovery amount

f — Payment frequency (for example, 2 for semiannual coupon bonds)

The default probability is:

`DefaultProbability` = 1 – `SurvivalProbability`

Version History

Introduced in R2017a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although bondDefaultBootstrap supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

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)

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. The clean price of a bond excludes any interest that has accrued since issue or the most recent coupon payment.

`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. Convexity is 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.

`[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 = datetime(1999,8,2);
Maturity = datetime(2004,6,15);
Period = 2;
Basis = 0;

[YearConvexity, PerConvexity] = bndconvp(Price,...
CouponRate,Settle, Maturity, Period, Basis)
```

```
YearConvexity = 3×1
```

```
   21.4447
   21.0363
   20.8951


PerConvexity = 3×1

   85.7788
   84.1454
   83.5803
```

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
datetime array | string array | date character vector

Settlement date for the certificate of deposit, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using a datetime array, string array, or date character vectors. The `Settle` date must be before the `Maturity` date.

To support existing code, `bndconvp` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Maturity` — Maturity date for certificate of deposit
datetime array | string array | date character vector

Maturity date for the certificate of deposit, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using a datetime array, string array, or date character vectors.

To support existing code, `bndconvp` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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 the comma-separated pair consisting of `'Period'` and a 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 the comma-separated pair consisting of `'Basis'` and a 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 2-16.

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 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

datetime array | string array | date character vector

Bond Issue date, specified as the comma-separated pair consisting of 'IssueDate' and a scalar or a NUMBONDS-by-1 or 1-by-NUMBONDS vector using a datetime array, string array, or date character vectors.

If you do not specify an IssueDate, the cash flow payment dates are determined from other inputs.

To support existing code, bndconvp also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

FirstCouponDate — Irregular or normal first coupon date

datetime array | string array | date character vector

Irregular or normal first coupon date, specified as the comma-separated pair consisting of 'FirstCouponDate' and a scalar or a NUMBONDS-by-1 or 1-by-NUMBONDS vector using a datetime array, string array, or date character vectors.

If you do not specify a FirstCouponDate, the cash flow payment dates are determined from other inputs.

To support existing code, bndconvp also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

LastCouponDate — Irregular or normal last coupon date

datetime array | string array | date character vector

Irregular or normal last coupon date, specified as the comma-separated pair consisting of 'LastCouponDate' and a scalar or a NUMBONDS-by-1 or 1-by-NUMBONDS vector using a datetime array, string array, or date character vectors.

If you do not specify a LastCouponDate, the cash flow payment dates are determined from other inputs.

To support existing code, bndconvp also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

StartDate — Forward starting date of payments

datetime array | string array | date character vector

Forward starting date of payments, specified as the comma-separated pair consisting of 'StartDate' and a scalar or a NUMBONDS-by-1 or 1-by-NUMBONDS vector using a datetime array,

string array, or date character vectors. 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.

To support existing code, `bndconvp` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

Face — Face value of bond
`100` (default) | numeric

Face value of the bond, specified as the comma-separated pair consisting of `'Face'` and 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 the comma-separated pair consisting of `'CompundingFrequency'` and 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 the comma-separated pair consisting of `'DiscountBasis'` and 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 2-16.

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 the comma-separated pair consisting of `'LastCouponInterest'` and 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.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although `bndconvp` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

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

`cfconv` | `bndconvy` | `bnddurp` | `bnddury` | `cfdur` | `datetime`

Topics

"Bond Portfolio for Hedging Duration and Convexity" on page 10-6
"Yield Conventions" on page 2-21

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 = datetime(1999,8,2);
Maturity = datetime(2004,6,15);
Period = 2;
Basis = 0;

[YearConvexity, PerConvexity]=bndconvy(Yield, CouponRate,...
Settle, Maturity, Period, Basis)
```

```
YearConvexity = 3×1

   21.4825
   21.0358
   20.8885
```

```
PerConvexity = 3×1

   85.9298
   84.1434
   83.5541
```

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
datetime array | string array | date character vector

Settlement date for the certificate of deposit, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using a datetime array, string array, or date character vectors. The `Settle` date must be before the `Maturity` date.

To support existing code, `bndconvy` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Maturity` — Maturity date for certificate of deposit
datetime array | string array | date character vector

Maturity date for the certificate of deposit, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using a datetime array, string array, or date character vectors.

To support existing code, `bndconvy` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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 the comma-separated pair consisting of `'Period'` and a 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 the comma-separated pair consisting of `'Basis'` and a 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 2-16.

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 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

datetime array | string array | date character vector

Bond Issue date, specified as the comma-separated pair consisting of `'IssueDate'` and a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using a datetime array, string array, or date character vectors.

If you do not specify an `IssueDate`, the cash flow payment dates are determined from other inputs.

To support existing code, `bndconvy` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

FirstCouponDate — Irregular or normal first coupon date

datetime array | string array | date character vector

Irregular or normal first coupon date, specified as the comma-separated pair consisting of `'FirstCouponDate'` and a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using a datetime array, string array, or date character vectors.

If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

To support existing code, `bndconvy` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

LastCouponDate — Irregular or normal last coupon date

datetime array | string array | date character vector

Irregular or normal last coupon date, specified as the comma-separated pair consisting of `'LastCouponDate'` and a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using a datetime array, string array, or date character vectors.

If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

To support existing code, `bndconvy` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

StartDate — Forward starting date of payments

datetime array | string array | date character vector

Forward starting date of payments, specified as the comma-separated pair consisting of `'StartDate'` and a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using a datetime array, string array, or date character vectors. 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.

To support existing code, `bndconvy` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

Face — Face value of bond
100 (default) | numeric

Face value of the bond, specified as the comma-separated pair consisting of 'Face' and 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 the comma-separated pair consisting of 'CompoundingFrequency' and 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 the comma-separated pair consisting of 'DiscountBasis' and 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 2-16.

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 the comma-separated pair consisting of `'LastCouponInterest'` and 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.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended
Not recommended starting in R2022b

Although `bndconvy` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

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

`cfconv` | `bndconvp` | `bnddurp` | `bnddury` | `cfdur` | `datetime`

Topics

"Bond Portfolio for Hedging Duration and Convexity" on page 10-6
"Yield Conventions" on page 2-21

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 = datetime(1999,8,2);
Maturity = datetime(2004,6,15);
Period = 2;
Basis = 0;

[ModDuration, YearDuration, PerDuration] = bnddurp(Price,...
CouponRate, Settle, Maturity, Period, Basis)
```

```
ModDuration = 3×1

    4.2400
    4.1925
    4.1759


YearDuration = 3×1

    4.3275
    4.3077
    4.3007


PerDuration = 3×1

    8.6549
    8.6154
    8.6014
```

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
datetime array | string array | date character vector

Settlement date for the certificate of deposit, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using a datetime array, string array, or date character vectors. The `Settle` date must be before the `Maturity` date.

To support existing code, `bnddurp` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Maturity` — Maturity date for certificate of deposit
datetime array | string array | date character vector

Maturity date for the certificate of deposit, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using a datetime array, string array, or date character vectors.

To support existing code, `bnddurp` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

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 the comma-separated pair consisting of 'Period' and a 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 the comma-separated pair consisting of 'Basis' and a 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 2-16.

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 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
datetime array | string array | date character vector

Bond Issue date, specified as the comma-separated pair consisting of 'IssueDate' and a scalar or a NUMBONDS-by-1 or 1-by-NUMBONDS vector using a datetime array, string array, or date character vectors.

If you do not specify an IssueDate, the cash flow payment dates are determined from other inputs.

To support existing code, bnddurp also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

FirstCouponDate — Irregular or normal first coupon date
datetime array | string array | date character vector

Irregular or normal first coupon date, specified as the comma-separated pair consisting of 'FirstCouponDate' and a scalar or a NUMBONDS-by-1 or 1-by-NUMBONDS vector using a datetime array, string array, or date character vectors.

If you do not specify a FirstCouponDate, the cash flow payment dates are determined from other inputs.

To support existing code, bnddurp also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

LastCouponDate — Irregular or normal last coupon date
datetime array | string array | date character vector

Irregular or normal last coupon date, specified as the comma-separated pair consisting of 'LastCouponDate' and a scalar or a NUMBONDS-by-1 or 1-by-NUMBONDS vector using a datetime array, string array, or date character vectors.

If you do not specify a LastCouponDate, the cash flow payment dates are determined from other inputs.

To support existing code, bnddurp also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

StartDate — Forward starting date of payments
datetime array | string array | date character vector

Forward starting date of payments, specified as the comma-separated pair consisting of 'StartDate' and a scalar or a NUMBONDS-by-1 or 1-by-NUMBONDS vector using a datetime array,

string array, or date character vectors. 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.

To support existing code, `bnddurp` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

Face — Face value of bond
`100` (default) | numeric

Face value of the bond, specified as the comma-separated pair consisting of `'Face'` and 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 the comma-separated pair consisting of `'CompoundingFrequency'` and 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 the comma-separated pair consisting of `'DiscountBasis'` and 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 2-16.

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 the comma-separated pair consisting of `'LastCouponInterest'` and 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.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although bnddurp supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

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 | bndconvp | bndkrdur | bnddury | datetime

Topics

“Bond Portfolio for Hedging Duration and Convexity” on page 10-6
“Yield Conventions” on page 2-21

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 = datetime(1999,8,2);
Maturity = datetime(2004,6,15);
Period = 2;
Basis = 0;

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

```
ModDuration = 3×1

    4.2444
    4.1924
    4.1751

YearDuration = 3×1

    4.3292
    4.3077
    4.3004

PerDuration = 3×1

    8.6585
    8.6154
    8.6007
```

Input Arguments

`Yield` — Yield to maturity on a semiannual basis
decimal

Yield to maturity on a semiannual basis, specified as a decimal 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
datetime array | string array | date character vector

Settlement date for the certificate of deposit, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using a datetime array, string array, or date character vectors. The `Settle` date must be before the `Maturity` date.

To support existing code, `bnddury` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Maturity` — Maturity date for certificate of deposit
datetime array | string array | date character vector

Maturity date for the certificate of deposit, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using a datetime array, string array, or date character vectors.

To support existing code, `bnddury` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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 the comma-separated pair consisting of `'Period'` and a 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 the comma-separated pair consisting of `'Basis'` and a 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 2-16.

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 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
datetime array | string array | date character vector

Bond Issue date, specified as the comma-separated pair consisting of 'IssueDate' and a scalar or a NUMBONDS-by-1 or 1-by-NUMBONDS vector using a datetime array, string array, or date character vectors.

If you do not specify an IssueDate, the cash flow payment dates are determined from other inputs.

To support existing code, bnddury also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

FirstCouponDate — Irregular or normal first coupon date
datetime array | string array | date character vector

Irregular or normal first coupon date, specified as the comma-separated pair consisting of 'FirstCouponDate' and a scalar or a NUMBONDS-by-1 or 1-by-NUMBONDS vector using a datetime array, string array, or date character vectors.

If you do not specify a FirstCouponDate, the cash flow payment dates are determined from other inputs.

To support existing code, bnddury also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

LastCouponDate — Irregular or normal last coupon date
datetime array | string array | date character vector

Irregular or normal last coupon date, specified as the comma-separated pair consisting of 'LastCouponDate' and a scalar or a NUMBONDS-by-1 or 1-by-NUMBONDS vector using a datetime array, string array, or date character vectors.

If you do not specify a LastCouponDate, the cash flow payment dates are determined from other inputs.

To support existing code, bnddury also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

StartDate — Forward starting date of payments
datetime array | string array | date character vector

Forward starting date of payments, specified as the comma-separated pair consisting of 'StartDate' and a scalar or a NUMBONDS-by-1 or 1-by-NUMBONDS vector using a datetime array,

string array, or date character vectors. 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.

To support existing code, bnddury also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

Face — Face value of bond
100 (default) | numeric

Face value of the bond, specified as the comma-separated pair consisting of 'Face' and 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 the comma-separated pair consisting of 'CompoundingFrequency' and 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 the comma-separated pair consisting of 'DiscountBasis' and 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 2-16.

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 the comma-separated pair consisting of `'LastCouponInterest'` and 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.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although `bnddury` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

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` | `bndconvp` | `bndkrdur` | `bnddurp` | `datetime`

Topics

"Bond Portfolio for Hedging Duration and Convexity" on page 10-6
"Yield Conventions" on page 2-21

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 = 1×4

    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'), ZeroRates);

krdur = bndkrdur(ZeroData,.0525,datetime('12/31/1998','Locale','en_US'),...
datetime('11/15/2028','Locale','en_US'),'KeyRates',[2 5 10 30])
```

```
krdur = 1×4

    0.2986    0.8791    4.1353    9.5814
```

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 a datetime array, string array, or date character vectors. The second column must be numeric data corresponding to the zero rates.

Data Types: double | datetime | char | string | 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

datetime scalar | string scalar | date character vector

Settlement date for all bonds and zero curve, specified as a scalar datetime, string, or date character vector. Settle must be the same settlement date for all the bonds and the zero curve.

To support existing code, bndkrdur also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

Maturity — Maturity date for bonds

datetime array | string array | date character vector

Maturity date for bonds, specified as a scalar or a NUMBONDS-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, bndkrdur also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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 the comma-separated pair consisting of `'InterpMethod'` and 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 the comma-separated pair consisting of `'ShiftValue'` and a 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 the comma-separated pair consisting of `'KeyRates'` and 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 the comma-separated pair consisting of `'CurveCompounding'` and 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 the comma-separated pair consisting of `'CurveBasis'` and 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 2-16.

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 the comma-separated pair consisting of `'Period'` and a 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 the comma-separated pair consisting of `'Basis'` and a 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 2-16.

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 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
datetime array | string array | date character vector

Bond Issue date, specified as the comma-separated pair consisting of `'IssueDate'` and a scalar or a `NUMBONDS`-by-`1` vector using a datetime array, string array, or date character vectors.

If you do not specify an `IssueDate`, the cash flow payment dates are determined from other inputs.

To support existing code, `bndkrdur` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`FirstCouponDate` — Irregular or normal first coupon date
datetime array | string array | date character vector

Irregular or normal first coupon date, specified as the comma-separated pair consisting of `'FirstCouponDate'` and a scalar or a `NUMBONDS`-by-`1` vector using a datetime array, string array, or date character vectors.

If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

To support existing code, `bndkrdur` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`LastCouponDate` — Irregular or normal last coupon date
datetime array | string array | date character vector

Irregular or normal last coupon date, specified as the comma-separated pair consisting of `'LastCouponDate'` and a scalar or a `NUMBONDS`-by-`1` vector using a datetime array, string array, or date character vectors.

If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

To support existing code, `bndkrdur` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`StartDate` — Forward starting date of payments

datetime array | string array | date character vector

Forward starting date of payments, specified as the comma-separated pair consisting of `'StartDate'` and a scalar or a `NUMBONDS`-by-`1` vector using a datetime array, string array, or date character vectors. 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.

To support existing code, `bndkrdur` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Face` — Face value of bond

`100` (default) | numeric

Face value of the bond, specified as the comma-separated pair consisting of `'Face'` and 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.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although bndkrdur supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

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

bndconvy | bndconvp | bnddury | bnddurp | datetime

Topics

"Bond Portfolio for Hedging Duration and Convexity" on page 10-6
"Yield Conventions" on page 2-21

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.

In addition, you can use the Financial Instruments Toolbox object framework with a `FixedBond` instrument to price a fixed bond.

`[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 = 3×1

  104.8106
   99.9951
   95.4384


AccruedInt = 3×1
```

```
    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 = 3×1

  104.8106
   99.9951
   95.4384


AccruedInt = 3×1

    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,datetime(2004,5,25),datetime(2005,4,21),'Period',1,'Basis',8, ...
'LastCouponInterest','simple')
```

```
ans = 103.4743
```

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
datetime array | string array | date character vector

Settlement date of the bond, specified as a scalar or a NUMBONDS-by-1 or 1-by-NUMBONDS vector using a datetime array, string array, or date character vectors. The Settle date must be before the Maturity date.

To support existing code, bndprice also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

Maturity — Maturity date of bond
datetime array | string array | date character vector

Maturity date of the bond, specified as a scalar or a NUMBONDS-by-1 or 1-by-NUMBONDS vector using a datetime array, string array, or date character vectors.

To support existing code, bndprice also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

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 the comma-separated pair consisting of 'Period' and a 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 the comma-separated pair consisting of `'Basis'` and a 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 2-16.

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 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
datetime array | string array | date character vector

Bond Issue date, specified as the comma-separated pair consisting of `'IssueDate'` and a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using a datetime array, string array, or date character vectors.

If you do not specify an `IssueDate`, the cash flow payment dates are determined from other inputs.

To support existing code, `bndprice` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

FirstCouponDate — Irregular or normal first coupon date

datetime array | string array | date character vector

Irregular or normal first coupon date, specified as the comma-separated pair consisting of 'FirstCouponDate' and a scalar or a NUMBONDS-by-1 or 1-by-NUMBONDS vector using a datetime array, string array, or date character vectors.

If you do not specify a FirstCouponDate, the cash flow payment dates are determined from other inputs.

To support existing code, bndprice also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

LastCouponDate — Irregular or normal last coupon date

datetime array | string array | date character vector

Irregular or normal last coupon date, specified as the comma-separated pair consisting of 'LastCouponDate' and a scalar or a NUMBONDS-by-1 or 1-by-NUMBONDS vector using a datetime array, string array, or date character vectors.

If you do not specify a LastCouponDate, the cash flow payment dates are determined from other inputs.

To support existing code, bndprice also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

StartDate — Forward starting date of payments

datetime array | string array | date character vector

Forward starting date of payments, specified as the comma-separated pair consisting of 'StartDate' and a scalar or a NUMBONDS-by-1 or 1-by-NUMBONDS vector using a datetime array, string array, or date character vectors.

If you do not specify a StartDate, the effective start date is the Settle date.

To support existing code, bndprice also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

Face — Face value of bond

100 (default) | numeric

Face value of the bond, specified as the comma-separated pair consisting of 'Face' and 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 the comma-separated pair consisting of 'CompoundingFrequency' and 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 the comma-separated pair consisting of 'DiscountBasis' and 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 2-16.

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 the comma-separated pair consisting of `'LastCouponInterest'` and 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.

More About

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.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although bndprice supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)

y =

        2021
```

There are no plans to remove support for serial date number inputs.

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 10-6
“Pricing Functions” on page 2-21
“Sensitivity of Bond Prices to Interest Rates” on page 10-2
“Bond Portfolio Optimization Using Portfolio Object” on page 10-30
“Yield Conventions” on page 2-21

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 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];
```

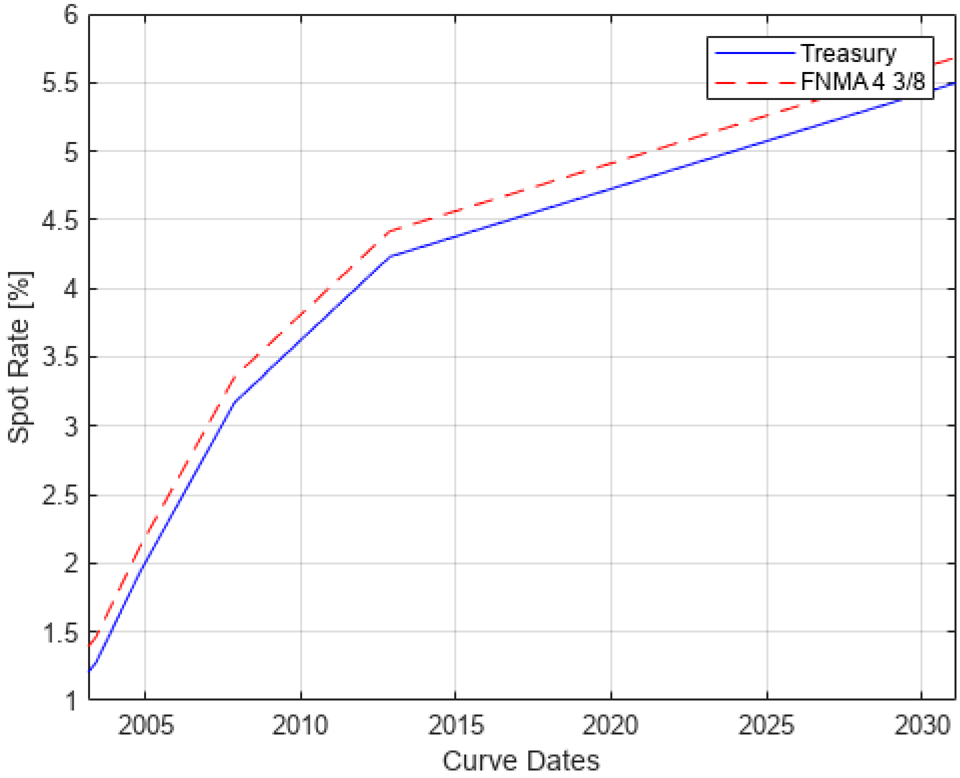
```
RefBonds = [RefPrices, RefMaturity, RefCpn];
Settle   = datetime('26-Nov-2002','Locale','en_US');
[ZeroRates, CurveDates] = zbtprice(RefBonds(:, 2:end), ...
RefPrices, Settle)
```

```
ZeroRates = 6×1

    0.0121
    0.0127
    0.0194
    0.0317
    0.0423
    0.0550


CurveDates = 6x1 datetime
   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(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;
```



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 = datetime(2004,1,1);
  Rates = [0.03840; 0.04512; 0.05086];
  EndDates = [datetime(2004,1,2) datenum('2-Jan-2004'); datetime(2005,1,2);...
               datetime(2006,1,2)];
  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 datetime array, string array, or date character vector. The second column is numerical data representing zero rates. For example:

  ```
  ZeroRates = [0.012067955808764;0.012730933424479;0.019360902068703;0.031704525214251;0.042306085224510;0.054987415342936];
  CurveDates = [731639;731730;732251;733361;735188;741854];
  Settle   = datetime(2002,11,26);
  Price    = 105.484;
  Coupon   = 0.04375;
  Maturity = datetime(2006,10,15);
  Period = 2;
  Basis  = 1;
  SpotInfo = table(datestr(CurveDates), ZeroRates);
  ```

Data Types: `double` | `string` | `char` | `datetime` | `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

datetime array | string array | date character vector

Settlement date of the bond, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using a datetime array, string array, or date character vectors. The `Settle` date must be before the `Maturity` date.

To support existing code, `bndspread` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Maturity` — Maturity date of bond

datetime array | string array | date character vector

Maturity date of the bond, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using a datetime array, string array, or date character vectors.

To support existing code, `bndspread` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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 the comma-separated pair consisting of `'Period'` and a 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 the comma-separated pair consisting of `'Basis'` and a 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 2-16.

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 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

datetime array | string array | date character vector

Bond Issue date, specified as the comma-separated pair consisting of 'IssueDate' and a scalar or a NUMBONDS-by-1 or 1-by-NUMBONDS vector using a datetime array, string array, or date character vectors.

If you do not specify an IssueDate, the cash flow payment dates are determined from other inputs.

To support existing code, bndspread also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

FirstCouponDate — Irregular or normal first coupon date

datetime array | string array | date character vector

Irregular or normal first coupon date, specified as the comma-separated pair consisting of 'FirstCouponDate' and a scalar or a NUMBONDS-by-1 or 1-by-NUMBONDS vector using a datetime array, string array, or date character vectors.

If you do not specify a FirstCouponDate, the cash flow payment dates are determined from other inputs.

To support existing code, bndspread also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

LastCouponDate — Irregular or normal last coupon date

datetime array | string array | date character vector

Irregular or normal last coupon date, specified as the comma-separated pair consisting of 'LastCouponDate' and a scalar or a NUMBONDS-by-1 or 1-by-NUMBONDS vector using a datetime array, string array, or date character vectors.

If you do not specify a LastCouponDate, the cash flow payment dates are determined from other inputs.

To support existing code, bndspread also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

StartDate — Forward starting date of payments

datetime array | string array | date character vector

Forward starting date of payments, specified as the comma-separated pair consisting of 'StartDate' and a scalar or a NUMBONDS-by-1 or 1-by-NUMBONDS vector using a datetime array,

string array, or date character vectors. 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.

To support existing code, `bndspread` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

Face — Face value of bond
`100` (default) | numeric

Face value of the bond, specified as the comma-separated pair consisting of `'Face'` and 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 the comma-separated pair consisting of `'CompoundingFrequency'` and 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 the comma-separated pair consisting of `'DiscountBasis'` and 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 2-16.

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 the comma-separated pair consisting of `'LastCouponInterest'` and 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.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended
Not recommended starting in R2022b

Although bndspread supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

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 | bndprice | datetime

Topics

"Bond Portfolio for Hedging Duration and Convexity" on page 10-6
"Pricing Functions" on page 2-21
"Sensitivity of Bond Prices to Interest Rates" on page 10-2
"Yield Conventions" on page 2-21

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 = 5×1

    0.0557
    0.0538
    0.0521
    0.0505
    0.0490


EffectiveRate = 5×1

    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
```

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

datetime array | string array | date character vector

Settlement date of the fixed-coupon bond, specified as scalar or a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors. 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`.

To support existing code, `bndtotalreturn` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Maturity` — Maturity date of fixed-coupon bond

datetime array | string array | date character vector

Maturity date of the fixed-coupon bond, specified as scalar or a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `bndtotalreturn` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[BondEquiv,EffectiveRate] = bndtotalreturn(Price,CouponRate,Settle,Maturity,ReinvestRate,'HorizonDate','15-Nov-2021')`

`HorizonDate` — Investment horizon date

`Maturity` date (default) | datetime array | string array | date character vector

Investment horizon date, specified as the comma-separated pair consisting of `'HorizonDate'` and a scalar or a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

If `HorizonDate` is unspecified, the total return is calculated to `Maturity`.

To support existing code, `bndtotalreturn` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`HorizonPrice` — Price at investment horizon date

calculated based on `ReinvestRate` (default) | numeric

Price at investment horizon date, specified as the comma-separated pair consisting of `'HorizonPrice'` and 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 the comma-separated pair consisting of `'Period'` and a 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 the comma-separated pair consisting of `'Basis'` and 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 2-16.

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 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
datetime array | string array | date character vector

Bond Issue date, specified as the comma-separated pair consisting of `'IssueDate'` and a scalar or a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

If you do not specify an `IssueDate`, the cash flow payment dates are determined from other inputs.

To support existing code, `bndtotalreturn` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`FirstCouponDate` — Irregular or normal first coupon date
datetime array | string array | date character vector

Irregular or normal first coupon date, specified as the comma-separated pair consisting of `'FirstCouponDate'` and a scalar or a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

To support existing code, `bndtotalreturn` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`LastCouponDate` — Irregular or normal last coupon date
datetime array | string array | date character vector

Irregular or normal last coupon date, specified as the comma-separated pair consisting of `'LastCouponDate'` and a scalar or a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

To support existing code, `bndtotalreturn` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`StartDate` — Forward starting date of payments
datetime array | string array | date character vector

Forward starting date of payments, specified as the comma-separated pair consisting of `'StartDate'` and a scalar or a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

If you do not specify a StartDate, the effective start date is the Settle date.

To support existing code, bndtotalreturn also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

Face — Face value of bond
100 (default) | numeric

Face value of the bond, specified as the comma-separated pair consisting of 'Face' and 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 the comma-separated pair consisting of 'CompoundingFrequency' and 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 the comma-separated pair consisting of 'DiscountBasis' and 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 2-16.

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.

Version History

Introduced in R2012b

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although bndtotalreturn supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)

y =

        2021
```

There are no plans to remove support for serial date number inputs.

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

`bndyield` | `bndprice` | `cfamounts` | `datetime`

Topics

"Bond Portfolio for Hedging Duration and Convexity" on page 10-6
"Pricing Functions" on page 2-21
"Sensitivity of Bond Prices to Interest Rates" on page 10-2
"Yield Conventions" on page 2-21

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 = 3×1

    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 = 3×1

    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 = datetime(2016,5,15);
Maturity = datetime(2026,2,2);
Period = 2;
Basis = 1;
format long

Yield = bndyield(Price,CouponRate,Settle,Maturity,Period,Basis)
```

```
Yield = 3×1

   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 = 3×1

   0.040780176658036
   0.034495592361619
```

```
    0.028565614029497
```

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

datetime array | string array | date character vector

Settlement date of the bond, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using a datetime array, string array, or date character vectors. The `Settle` date must be before the `Maturity` date.

To support existing code, `bndyield` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Maturity` — Maturity date of bond

datetime array | string array | date character vector

Maturity date of the bond, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using a datetime array, string array, or date character vectors.

To support existing code, `bndyield` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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 the comma-separated pair consisting of 'Period' and a 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 the comma-separated pair consisting of 'Basis' and a 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 2-16.

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 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

datetime array | string array | date character vector

Bond Issue date, specified as the comma-separated pair consisting of `'IssueDate'` and a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using a datetime array, string array, or date character vectors.

If you do not specify an `IssueDate`, the cash flow payment dates are determined from other inputs.

To support existing code, `bndyield` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`FirstCouponDate` — Irregular or normal first coupon date

datetime array | string array | date character vector

Irregular or normal first coupon date, specified as the comma-separated pair consisting of `'FirstCouponDate'` and a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using a datetime array, string array, or date character vectors.

If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

To support existing code, `bndyield` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`LastCouponDate` — Irregular or normal last coupon date

datetime array | string array | date character vector

Irregular or normal last coupon date, specified as the comma-separated pair consisting of `'LastCouponDate'` and a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using a datetime array, string array, or date character vectors.

If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

To support existing code, `bndyield` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`StartDate` — Forward starting date of payments

datetime array | string array | date character vector

Forward starting date of payments, specified as the comma-separated pair consisting of `'StartDate'` and a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using a datetime array, string array, or date character vectors.

If you do not specify a `StartDate`, the effective start date is the `Settle` date.

To support existing code, `bndyield` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Face` — Face value of bond

`100` (default) | numeric

Face value of the bond, specified as the comma-separated pair consisting of `'Face'` and 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 the comma-separated pair consisting of `'CompoundingFrequency'` and 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 the comma-separated pair consisting of `'DiscountBasis'` and 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 2-16.

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 the comma-separated pair consisting of 'LastCouponInterest' and 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.

More About

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.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although `bndyield` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

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 10-6
"Pricing Functions" on page 2-21
"Sensitivity of Bond Prices to Interest Rates" on page 10-2

“Yield Conventions” on page 2-21

bollinger

Time series Bollinger band

Syntax

```
[middle,upper,lower] = bollinger(Data)
[middle,upper,lower] = bollinger( ___ ,Name,Value)
```

Description

`[middle,upper,lower] = bollinger(Data)` calculates the middle, upper, and lower bands that make up the Bollinger bands from a series of data. A Bollinger band chart plots actual asset data along with three other bands of data: the upper band that is two standard deviations above a user-specified moving average; the lower band that is two standard deviations below that moving average; and the middle band that is the moving average itself.

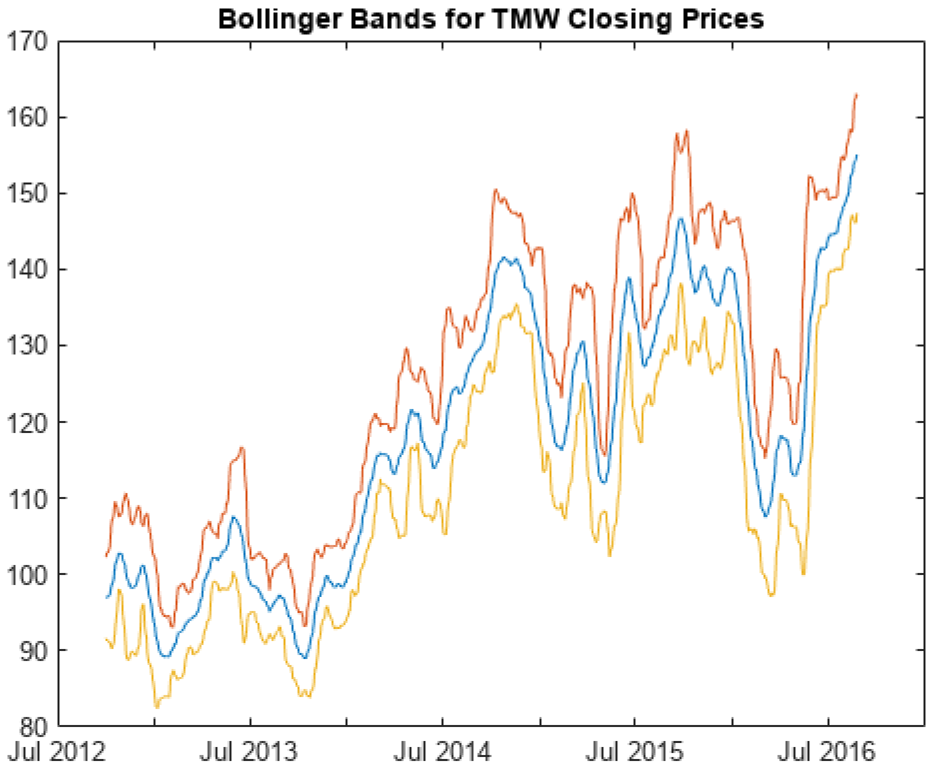
`[middle,upper,lower] = bollinger( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Calculate the Bollinger Bands for a Data Series for a Stock

Load the file `SimulatedStock.mat`, which provides a timetable (`TMW`) for financial data for TMW stock.

```
load SimulatedStock.mat
[middle,upper,lower]= bollinger(TMW);
CloseBolling = [middle.Close, upper.Close,...
lower.Close];
plot(middle.Time,CloseBolling)
title('Bollinger Bands for TMW Closing Prices')
```



Input Arguments

`Data` — Data for market prices

matrix | table | timetable

Data for market prices, specified as a matrix, table, or timetable. For matrix input, `Data` must be column-oriented.

Data Types: `double` | `table` | `timetable`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[middle,upper,lower] = bollinger(TMW_CLOSE,'WindowSize',10,'Type',1)`

`WindowSize` — Number of observations of input series to include in moving average in periods

`10` (default) | positive integer

Number of observations of the input series to include in the moving average in periods, specified as the comma-separated pair consisting of `'WindowSize'` and a scalar positive integer.

Data Types: `double`

Type — Type of moving average to compute

`0` (simple) (default) | integer with value `0` or `1`

Type of moving average to compute, specified as the comma-separated pair consisting of `'Type'` and a scalar integer with a value of `0` (simple) or `1` (linear).

Data Types: `double`

`NumStd` — Number of standard deviations for the upper and lower bounds

2 (default) | positive integer

Number of standard deviations for the upper and lower bounds, specified as the comma-separated pair consisting of `'NumStd'` and a scalar positive integer.

Data Types: `double`

Output Arguments

`middle` — Moving average series representing the middle band

matrix | table | timetable

Moving average series representing the middle band, returned with the same number of rows (`M`) and the same type (matrix, table, or timetable) as the input `Data`.

`upper` — Moving average series representing the upper band

matrix | table | timetable

Moving average series representing the upper band, returned with the same number of rows (`M`) and the same type (matrix, table, or timetable) as the input `Data`.

`lower` — Moving average series representing the lower band

matrix | table | timetable

Moving average series representing the lower band, returned with the same number of rows (`M`) and the same type (matrix, table, or timetable) as the input `Data`.

Version History

Introduced before R2006a

R2023a: `fints` support removed for `Data` input argument

Behavior changed in R2023a

`fints` object support for the `Data` input argument is removed.

R2022b: Support for negative price data

Behavior changed in R2022b

The `Data` input accepts negative prices.

References

[1] Achelis, S. B. *Technical Analysis from A to Z.* Second Edition. McGraw-Hill, 1995, pp. 72–74.

See Also

`timetable` | `table` | `movavg`

Topics

"Use Timetables in Finance" on page 11-7
"Convert Financial Time Series Objects (fints) to Timetables" on page 11-2

boxcox

Box-Cox transformation

Syntax

```
[transdat,lambda] = boxcox(data)
transdat = boxcox(lambda,data)
```

Description

`[transdat,lambda] = boxcox(data)` transforms the data vector `data` using the Box-Cox transformation method into `transdat`. It also estimates the transformation parameter λ.

`transdat = boxcox(lambda,data)` 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 Vector of Data

Use `boxcox` to transform the data series contained in a vector of data into another set of data series with relatively normal distributions.

Load the `SimulatedStock.mat` data file.

```
load SimulatedStock.mat
```

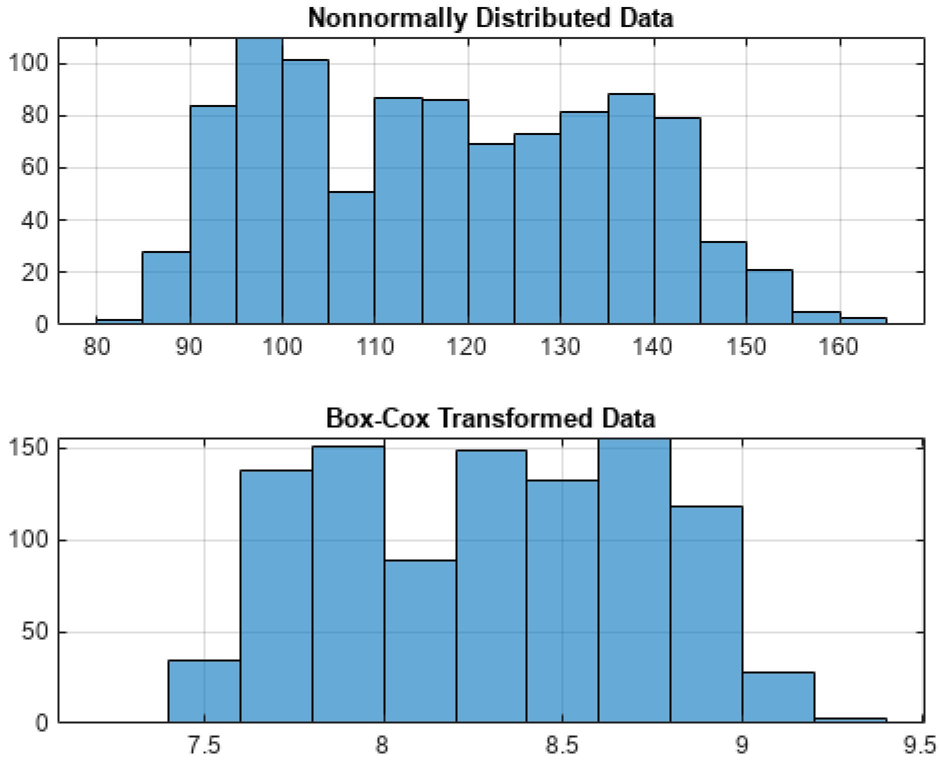
Transform the nonnormally distributed filled data series `TMW_CLOSE` into a normally distributed one using Box-Cox transformation.

```
 [Xbc, lambdabc] = boxcox(TMW_CLOSE)
```

```
Xbc = 1000×1

    7.8756
    7.8805
    7.9173
    7.8557
    7.8245
    7.7844
    7.7811
    7.8029
    7.8015
    7.7229
      ⋮

lambdabc = 0.2151
```

Compare the result of the `TMW_CLOSE` data series with a normal (Gaussian) probability distribution function and the nonnormally distributed `TMW_CLOSE`.

```
subplot(2, 1, 1);
histogram(TMW_CLOSE);
grid; title('Nonnormally Distributed Data');
subplot(2, 1, 2);
histogram(Xbc);
grid; title('Box-Cox Transformed Data');
```



The bar chart on the top represents the probability distribution function of the data series, TMW_CLOSE, which is the original data series. 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 close to normal (Gaussian). When you examine the contents of the resulting object Xbc, you find an identical object to the original object TMW_CLOSE but the contents are the transformed data series.

Input Arguments

data — Data
positive column vector

Data, specified as a positive column vector.

Data Types: double

lambda — Lambda
numeric | structure

Lambda, specified as a scalar numeric or structure.

If the input `data` is a vector, `lambda` is a scalar. If the input is a financial time series object (`tsobj`), `lambda` is a structure with fields similar to the components of the object. For example, if `tsobj` contains series names `Open` and `Close`, `lambda` has fields `lambda.Open` and `lambda.Close`.

Data Types: `double` | `struct`

Output Arguments

`transdat` — Data Box-Cox transformation
vector

Data Box-Cox transformation, returned as a vector.

`lambda` — Lambda transformation parameter
numeric

Lambda transformation parameter, returned as a numeric.

More About

Box Cox Transformation

`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`.

Version History

Introduced before R2006a

R2023a: `fints` support for `tsobj` input argument and `transfts` output argument are removed
Behavior changed in R2023a

The `tsobj` input argument that supports a `fints` object is and the `transfts` output argument are removed from `boxcox`.

See Also

`fminsearch`

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 strings or date character vectors, `Busday` is returned as a serial date number. Use the function `datestr` to convert serial date numbers to formatted date character vectors.

However, if either `Date` or `Holiday` are datetime arrays, `Busday` is returned as a datetime array.

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
```

Also, you can bypass a holiday.

```
busdate(datetime(2022,11,23),1,NaT)
```

```
ans = datetime
   24-Nov-2022
```

Input Arguments

Date — Reference business date

datetime array | string array | date character vector

Reference business date, specified as a scalar, vector, or matrix using a datetime array, string array, or date character vectors.

To support existing code, `busdate` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `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) | datetime array | string array | date character vector

Holidays and nontrading-day dates, specified as vector using a datetime array, string array, or date character vectors.

All dates in `Holiday` must be the same format: either datetimes, strings, or date character vectors.

To support existing code, `busdate` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `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`. `Busday`.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although `busdate` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`holidays` | `datetime`

Topics

"Handle and Convert Dates" on page 2-2
"Trading Calendars User Interface" on page 12-2

busdays

Business days for given period

Syntax

```
bdates = busdays(sdate,edate)
bdates = busdays( ___ ,bdmode,Holiday)
```

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,Holiday)` 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 `Holiday` 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
   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).

Input Arguments

sdate — Start date

datetime scalar | string scalar | date character vector

Start date, specified as a scalar datetime, string, or date character vector.

To support existing code, busdays also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

edate — End date

datetime scalar | string scalar | date character vector

End date, specified as a scalar datetime, string, or date character vector.

To support existing code, busdays also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | 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

Holiday — Holiday and nontrading-day dates

if Holiday is [ ] holiday dates are based on United States holidays specified by holidays (default) | datetime array | string array | date character vector

Holiday and nontrading-day dates, specified as a vector using a datetime array, string array, or date character vectors. If you specify Holiday, you must also supply the frequency bdmode. Using a Holiday value of NaN uses a holiday list that has no dates.

To support existing code, busdays also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

Output Arguments

bdates — Business days

column vector

Business days, returned as a column vector of business dates, in datetime format (if sdate, edate, or Holiday are in datetime format). Business dates can exist before and/or after the specified sdate and edate.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although busdays supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)

y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

isbusday | holidays | datetime

Topics

"Handle and Convert Dates" on page 2-2
"Trading Calendars User Interface" on page 12-2

candle

Candlestick chart

Note `candle` is updated to accept data input as a matrix, `timetable`, or `table`.

Syntax

```
candle(Data)
candle(Data,Color)
h = candle(ax___)
```

Description

`candle(Data)` plots a candlestick chart from a series of opening, high, low, and closing prices of a security. If the closing price is greater than the opening price, the body (the region between the open and close price) is unfilled; otherwise the body is filled.

`candle(Data,Color)` adds an optional argument for `Color`.

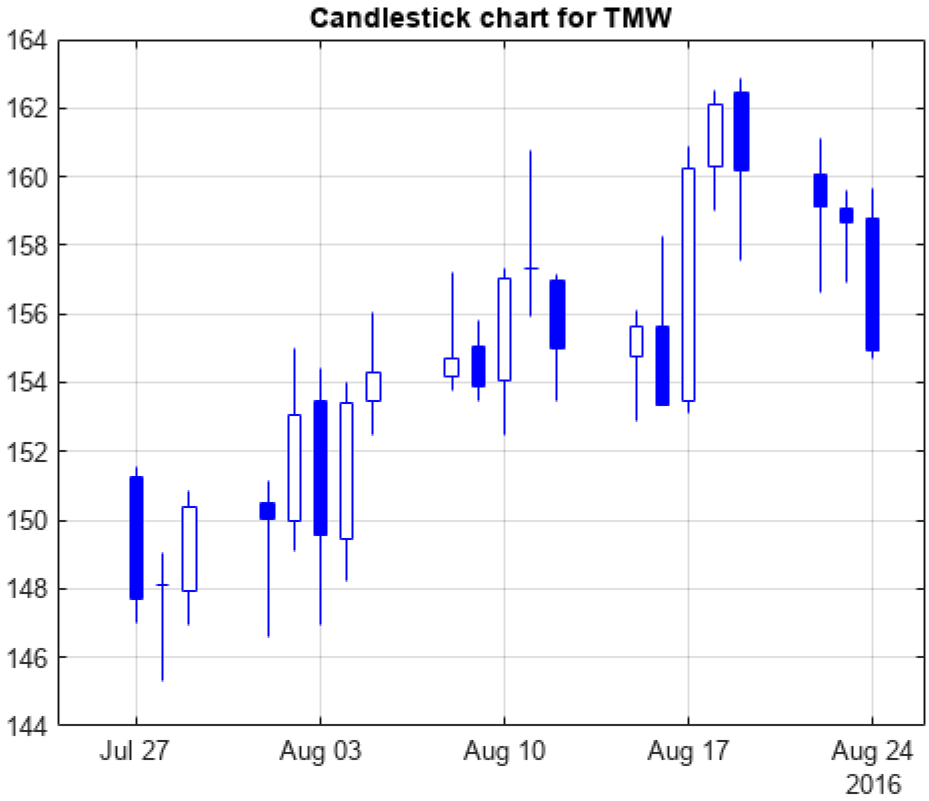
`h = candle(ax___)` adds an optional argument for `ax`.

Examples

Generate a Candlestick Chart for a Data Series for a Stock

Load the file `SimulatedStock.mat`, which provides a timetable (`TMW`) for financial data for TMW stock. This is a candlestick chart with blue candles, for the most recent 21 days in SimulatedStock`.mat`.

```
load SimulatedStock.mat;
candle(TMW(end-20:end,:),'b');
title('Candlestick chart for TMW')
```



Input Arguments

`Data` — Data for opening, high, low, and closing prices

matrix | table | timetable

Data for opening, high, low, and closing prices, specified as a matrix, table, or timetable. For matrix input, `Data` is an `M`-by-4 matrix of opening, high, low, and closing prices stored in the corresponding columns. Timetables and tables with `M` rows must contain variables named `'Open'`, `'High'`, `'Low'`, and `'Close'` (case insensitive).

Data Types: `double` | `table` | `timetable`

`Color` — (Optional) Three element color vector

background color of figure window (default) | color vector `[R G B]` | string

Three element color vector, specified as a `[R G B]` color vector or a string specifying the color name. The default color differs depending on the background color of the figure window.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.

- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`"red"`	`"r"`	`[1 0 0]`	`"#FF0000"`	🟥
`"green"`	`"g"`	`[0 1 0]`	`"#00FF00"`	🟩
`"blue"`	`"b"`	`[0 0 1]`	`"#0000FF"`	🟦
`"cyan"`	`"c"`	`[0 1 1]`	`"#00FFFF"`	🟦
`"magenta"`	`"m"`	`[1 0 1]`	`"#FF00FF"`	🟪
`"yellow"`	`"y"`	`[1 1 0]`	`"#FFFF00"`	🟨
`"black"`	`"k"`	`[0 0 0]`	`"#000000"`	⬛
`"white"`	`"w"`	`[1 1 1]`	`"#FFFFFF"`	⬜

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`"#0072BD"`	🟦
`[0.8500 0.3250 0.0980]`	`"#D95319"`	🟧
`[0.9290 0.6940 0.1250]`	`"#EDB120"`	🟨
`[0.4940 0.1840 0.5560]`	`"#7E2F8E"`	🟪
`[0.4660 0.6740 0.1880]`	`"#77AC30"`	🟩
`[0.3010 0.7450 0.9330]`	`"#4DBEEE"`	🟦
`[0.6350 0.0780 0.1840]`	`"#A2142F"`	🟥

Data Types: `double` | `string`

ax — Valid axis object

current axes (`ax = gca`) (default) | axes object

(Optional) Valid axis object, specified as an axes object. The candle plot is created in the axes specified by `ax` instead of in the current axes (`ax = gca`). The option `ax` can precede any of the input argument combinations.

Data Types: `object`

Output Arguments

h — Graphic handle of the figure

handle object

Graphic handle of the figure, returned as a handle object.

Version History

Introduced before R2006a

R2022b: Support for negative price data

Behavior changed in R2022b

The `Data` input accepts negative prices.

See Also

`timetable` | `table` | `highlow` | `movavg` | `pointfig` | `kagi` | `linebreak` | `priceandvol` | `renko` | `volarea`

Topics

"Use Timetables in Finance" on page 11-7
"Convert Financial Time Series Objects (fints) to Timetables" on page 11-2

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
```

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
datetime array | string array | date character vector

Settlement date for the certificate of deposit, specified as a scalar or a NCDS-by-1 or 1-by-NCDS vector using a datetime array, string array, or date character vectors. The Settle date must be before the Maturity date.

To support existing code, cdai also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

Maturity — Maturity date for certificate of deposit
datetime array | string array | date character vector

Maturity date for the certificate of deposit, specified as a scalar or a NCDS-by-1 or 1-by-NCDS vector using a datetime array, string array, or date character vectors.

To support existing code, cdai also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

IssueDate — Issue date for certificate of deposit
datetime array | string array | date character vector

Issue date for the certificate of deposit, specified as a scalar or a NCDS-by-1 or 1-by-NCDS vector using a datetime array, string array, or date character vectors.

To support existing code, cdai also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | 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 2-16.

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.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although `cdai` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`accrfrac` | `bndyield` | `stepcpnyield` | `tbillyield` | `cdprice` | `cdyield` | `zeroyield` | `datetime`

Topics

“Coupon Date Calculations” on page 2-20
“Yield Conventions” on page 2-21

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
```

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

datetime array | string array | date character vector

Settlement date for the certificate of deposit, specified as a scalar or a `NCDS`-by-`1` or `1`-by-`NCDS` vector using a datetime array, string array, or date character vectors. The `Settle` date must be before the `Maturity` date.

To support existing code, `cdprice` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Maturity` — Maturity date for certificate of deposit

datetime array | string array | date character vector

Maturity date for the certificate of deposit, specified as a scalar or a `NCDS`-by-`1` or `1`-by-`NCDS` vector using a datetime array, string array, or date character vectors.

To support existing code, `cdprice` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`IssueDate` — Issue date for certificate of deposit

datetime array | string array | date character vector

Issue date for the certificate of deposit, specified as a scalar or a `NCDS`-by-`1` or `1`-by-`NCDS` vector using a datetime array, string array, or date character vectors.

To support existing code, `cdprice` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `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 2-16.

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.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended
Not recommended starting in R2022b

Although `cdprice` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`bndprice` | `cdai` | `cdyield` | `stepcpnprice` | `tbillprice` | `datetime`

Topics

"Coupon Date Calculations" on page 2-20
"Yield Conventions" on page 2-21

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.

Note Alternatively, you can use the Financial Instruments Toolbox `defprobstrip` function to bootstrap a `defprobcurve` object from market CDS instruments. For more information, see “Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments” (Financial Instruments Toolbox).

`[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 = datetime(2009,7,17); % 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 = 5×2
10^5 ×

   7.343360000000000   0.000000233427859
   7.347010000000000   0.000000575839968
   7.350670000000000   0.000001021397017
   7.357970000000000   0.000002064539982
   7.365280000000000   0.000003234110940


HazData = 5×2
10^5 ×

   7.343360000000000   0.000000232959886
   7.347010000000000   0.000000352000512
   7.350670000000000   0.000000476383354
   7.357970000000000   0.000000609055766
   7.365280000000000   0.000000785241515
```

Input Arguments

ZeroData — Zero rate data

vector | IRDataCurve object

Zero rate data, specified as a M-by-2 vector of dates, using a serial date number format, 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 | object

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

datetime scalar | string scalar | date character vector

Settlement date, specified as a scalar datetime, string, or date character vector. The Settle date must be earlier than or equal to the dates in MarketData

To support existing code, cdsbootstrap also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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 the comma-separated pair consisting of `'RecoveryDate'` and 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 the comma-separated pair consisting of `'Period'` and 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 the comma-separated pair consisting of `'Basis'` and 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 2-16.

Data Types: `double`

`BusinessDayConvention` — Business day conventions

`'actual'` (default) | character vector

Business day conventions, specified as the comma-separated pair consisting of `'BusinessDayConvention'` and 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 the comma-separated pair consisting of `'PayAccruedPremium'` and 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 the comma-separated pair consisting of `'TimeStep'` and 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 as the comma-separated pair consisting of `'ZeroCompounding'` and an integer with 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, specified as the comma-separated pair consisting of 'ZeroBasis' and an integer with values that are identical to Basis.

Data Types: double

ProbDates — Dates for probability data

column of dates in MarketData (default) | datetime array | string array | date character vector

Dates for probability data, specified as the comma-separated pair consisting of 'ProbDates' and a P-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, cdsbootstrap also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

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 $0 \leq t \leq t_1$

λ_2, for $t_1 < t \leq t_2$

λ_3, for $t_2 < t$

then the survival function is given by *Q(t)* =

$e^{-\lambda 1t}$, for $0 \leq t \leq t_1$

$e^{-\lambda 1t - \lambda 2(t - t1)}$, for $t_1 < t \leq t_2$

$e^{-\lambda 1t1 - \lambda 2(t2 - t1) - \lambda 3(t - t2)}$, for $t_2 < t$

Given *n* market dates $t_1,\ldots,t_n$ and corresponding market CDS spreads $S_1,\ldots,S_n$, `cdsbootstrap` calibrates the parameters $\lambda_1,\ldots,\lambda_n$ and evaluates *PD(t)* on the market dates, or an optional user-defined set of dates.

Version History

Introduced in R2010b

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although `cdsbootstrap` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

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

`cdsspread` | `cdsprice` | `cdsrpv01` | `IRDataCurve` | `defprobstrip`

Topics

"Bootstrapping a Default Probability Curve" on page 8-98
"Bootstrapping from Inverted Market Curves" on page 8-108
"Credit Default Swap (CDS)" on page 8-97

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.

Note Alternatively, you can use the Financial Instruments Toolbox CDS object to price credit default swaps. For more information, see "Get Started with Workflows Using Object-Based Framework for Pricing Financial Instruments" (Financial Instruments Toolbox).

`[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 = datetime(2010,9,20);
ContractSpread = 135;

[Price,AccPrem] = cdsprice(ZeroData,ProbData,Settle,Maturity,ContractSpread)
```

```
Price = 1.5461e+04

AccPrem = 10500
```

Input Arguments

ZeroData — Zero rate data

vector | IRDataCurve object

Zero rate data, specified as a M-by-2 vector of dates, using a serial date number format, 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 | object

ProbData — Default probability values

matrix

Default probability values, specified as a P-by-2 matrix with dates, using a serial date number format, and the corresponding cumulative default probability values.

Data Types: double

Settle — Settlement date

datetime scalar | string scalar | date character vector

Settlement date, specified as a scalar datetime, string, or date character vector. The Settle date must be earlier than or equal to the dates in Maturity.

To support existing code, cdsprice also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

Maturity — Maturity date

datetime array | string array | date character vector

Maturity date, specified as a N-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, cdsprice also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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,'BusinessDayConvention','previous')`

`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, 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 the comma-separated pair consisting of `'Period'` and 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 the comma-separated pair consisting of `'Basis'` and 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 2-16.

Data Types: `double`

`BusinessDayConvention` — Business day conventions

`actual` (default) | character vector

Business day conventions, specified as the comma-separated pair consisting of `'BusinessDayConvention'` and 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 the comma-separated pair consisting of `'PayAccruedPremium'` and 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 the comma-separated pair consisting of `'Notional'` and 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 the comma-separated pair consisting of `'TimeStep'` and 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 as the comma-separated pair consisting of 'ZeroCmpounding' and an integer with 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, specified as the comma-separated pair consisting of 'ZeroBasis' and an integer with values that 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 — Accrued premiums

vector

Accrued 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.

More About

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:

$$RPV01 = \sum_{j=1}^{N} Z(tj)\Delta(tj-1, tj, B)Q(tj)$$

when no accrued premiums are paid upon default, and it can be approximated by

$$RPV01 \approx \frac{1}{2}\sum_{j=1}^{N} Z(tj)\Delta(tj-1, tj, B)(Q(tj-1)+Q(tj))$$

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:

`MtM` = `Notional` $(S_0 - S_C)$`RPV01`

This assumes a long position from the protection standpoint (protection was bought). For short positions, the sign is reversed.

Version History

Introduced in R2010b

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although cdsprice supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

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

cdsbootstrap | cdsspread | cdsoptprice | IRDataCurve

Topics

“Finding Breakeven Spread for New CDS Contract” on page 8-101
“Valuing an Existing CDS Contract” on page 8-104
“Converting from Running to Upfront” on page 8-106
“Bootstrapping a Default Probability Curve from Credit Default Swaps” (Financial Instruments Toolbox)
“Credit Default Swap (CDS)” on page 8-97

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 = datetime(2010,9,20);

Spread = cdsspread(ZeroData,ProbData,Settle,Maturity)
```

```
Spread = 148.2705
```

Input Arguments

ZeroData — Zero rate data

vector | `IRDataCurve` object

Zero rate data, specified as a `M`-by-`2` vector of dates, using a serial date number format, 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` | `object`

`ProbData` — Default probability values
matrix

Default probability values, specified as a `P`-by-2 matrix with dates, using a serial date number format, and corresponding cumulative default probability values.

Data Types: `double`

`Settle` — Settlement date
datetime scalar | string scalar | date character vector

Settlement date, specified as a scalar datetime, string, or date character vector. The `Settle` date must be earlier than or equal to the dates in `Maturity`.

To support existing code, `cdsspread` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Maturity` — Maturity date
datetime array | string array | date character vector

Maturity date, specified as a `N`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `cdsspread` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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,'BusinessDayConvention','previous')`

`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, 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 the comma-separated pair consisting of `'Period'` and 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 the comma-separated pair consisting of `'Basis'` and 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 2-16.

Data Types: `double`

`BusinessDayConvention` — Business day conventions

`actual` (default) | character vector

Business day conventions, specified as the comma-separated pair consisting of `'BusinessDayConvention'` and 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 the comma-separated pair consisting of 'PayAccruedPremium' and 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 the comma-separated pair consisting of 'TimeStep' and 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 as the comma-separated pair consisting of 'ZeroCompounding' and an integer with 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, specified as the comma-separated pair consisting of 'ZeroBasis' and a positive integer with values that 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.

More About

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:

$$RPV01 = \sum_{j=1}^{N} Z(tj)\Delta(tj-1, tj, B)Q(tj)$$

when no accrued premiums are paid upon default, and it can be approximated by

$$RPV01 \approx \frac{1}{2}\sum_{j=1}^{N} Z(tj)\Delta(tj-1, tj, B)(Q(tj-1) + Q(tj))$$

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 $\tau_0 = 0, \tau_1, \ldots, \tau_M = T$.

A breakeven spread S_0 makes the value of the premium and protection legs equal. It follows that:

$$S0 = \frac{ProtectionLeg}{RPV01}$$

Version History

Introduced in R2010b

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although `cdsspread` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

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

`cdsbootstrap` | `cdsprice` | `IRDataCurve`

Topics

"Finding Breakeven Spread for New CDS Contract" on page 8-101
"Valuing an Existing CDS Contract" on page 8-104
"Converting from Running to Upfront" on page 8-106
"First-to-Default Swaps" (Financial Instruments Toolbox)
"Pricing a CDS Index Option" (Financial Instruments Toolbox)
"Credit Default Swap (CDS)" on page 8-97

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 = datetime(2010,9,20);

RPV01 = cdsrpv01(ZeroData,ProbData,Settle,Maturity)
```

```
RPV01 = 1.1651
```

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, using a serial date number format, and zero rates or the `IRDataCurve` object for zero rates. For more information on an `IRDataCurve` object, see “Creating an IRDataCurve Object” (Financial Instruments Toolbox).

Data Types: `object` | `double`

`ProbData` — Dates and default probabilities

vector of dates and default probabilities

Dates and default probabilities, specified by a `P`-by-`2` array, where dates use a serial date number format.

Data Types: `double`

`Settle` — Settlement date

datetime scalar | string scalar | date character vector

Settlement date, specified by a scalar datetime, string, or date character vector. This must be earlier than or equal to the dates in `Maturity`.

To support existing code, `cdsrpv01` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Maturity` — CDS maturity date

datetime array | string array | date character vector | serial date number

CDS maturity date, specified by an `N`-by-`1` vector using a datetime array, string array, or date character vectors. The CDS premium payment dates occur at regular intervals, and the last payment occurs on these maturity dates.

To support existing code, `cdsrpv01` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `RPV01 = cdsrpv01(ZeroData,ProbData,Settle,Maturity,'Period',1,'StartDate','20-Sep-2010','Basis',1, 'BusinessDayConvention',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 as the comma-separated pair consisting of `'Period'` and 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) | datetime array | string array | date character vector

Dates when the CDS premium leg actually starts, specified as the comma-separated pair consisting of 'StartDate' and an N-by-1 vector using a datetime array, string array, 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.

To support existing code, cdsrpv01 also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

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 the comma-separated pair consisting of 'Basis' and 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 2-16.

Data Types: double

BusinessDayConvention — Business day conventions

actual (default) | character vector | cell array of character vectors

Business day conventions, specified as the comma-separated pair consisting of 'BusinessDayConvention' and 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 the comma-separated pair consisting of `'CleanRPV01'` and a `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 the comma-separated pair consisting of `'PayAccruedPremium'` and 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 as the comma-separated pair consisting of `'ZeroCompounding'` and an integer with 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 the comma-separated pair consisting of `'ZeroBasis'` and 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 2-16.

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.

More About

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.

Version History

Introduced in R2013b

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although `cdsrpv01` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

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.

[4] O'Kane, D. *Modelling Single-name and Multi-name Credit Derivatives.* Wiley Finance, 2008.

See Also

`cdsbootstrap` | `cdsspread` | `cdsprice` | `cdsoptprice` | `IRDataCurve`

Topics

“Pricing a CDS Index Option” (Financial Instruments Toolbox)
“Credit Default Swap Option” (Financial Instruments Toolbox)

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

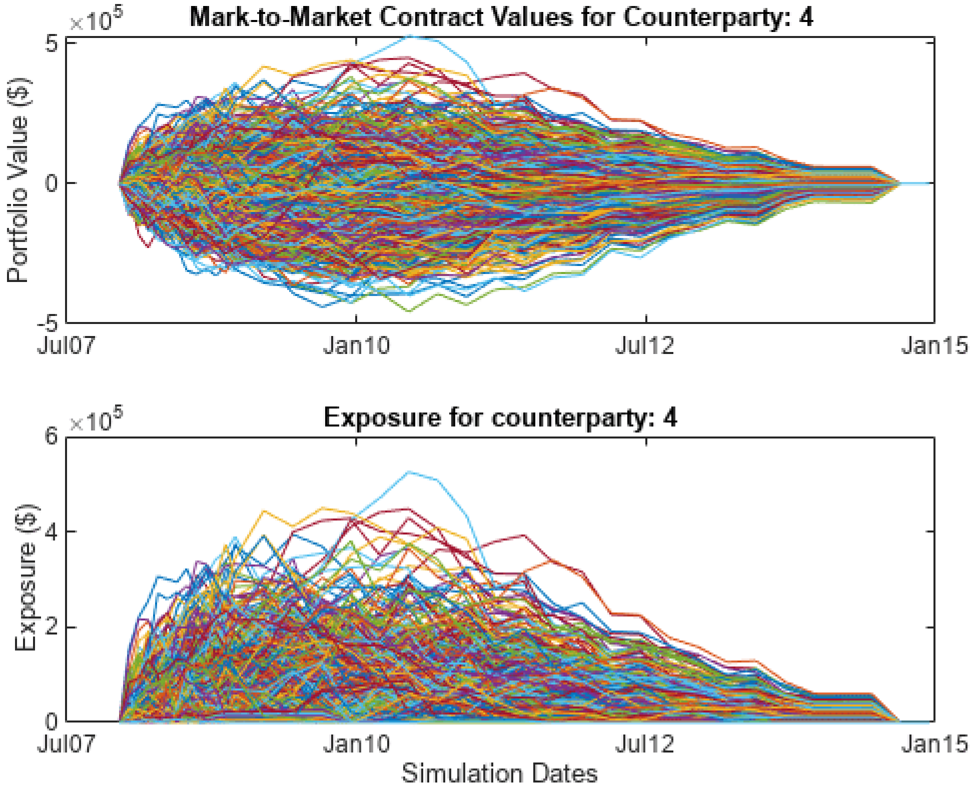
View Contract Values and Exposures Over Time for a Particular Counterparty

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')
```



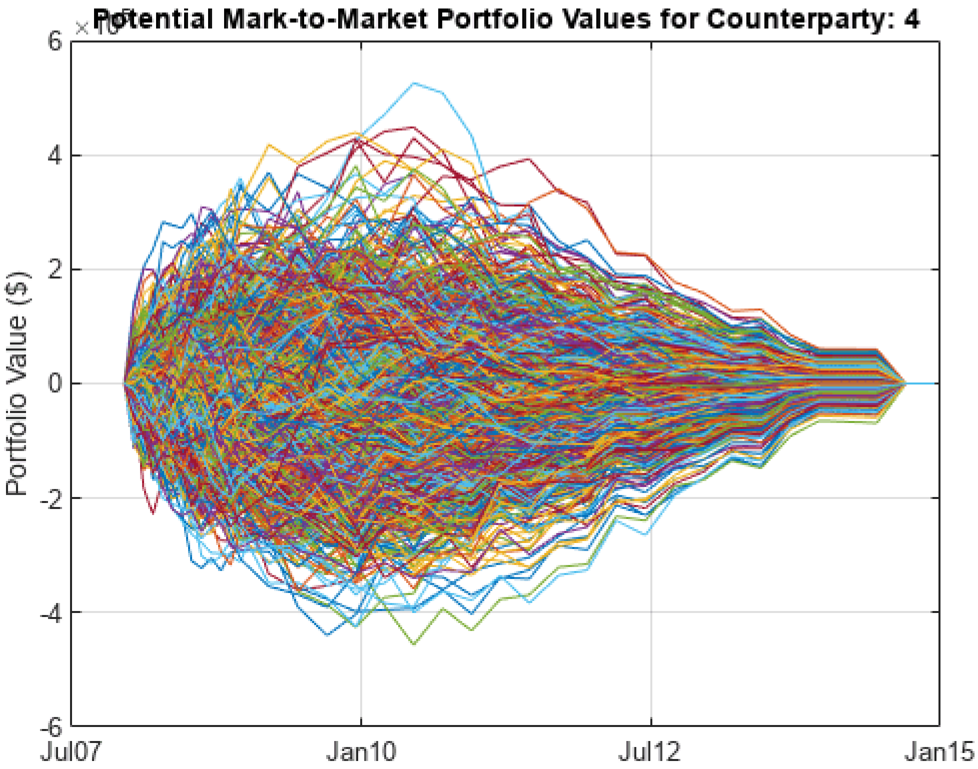
Compute the Credit Exposure and Determine the Incremental Exposure for a New Trade

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 ($)')
```

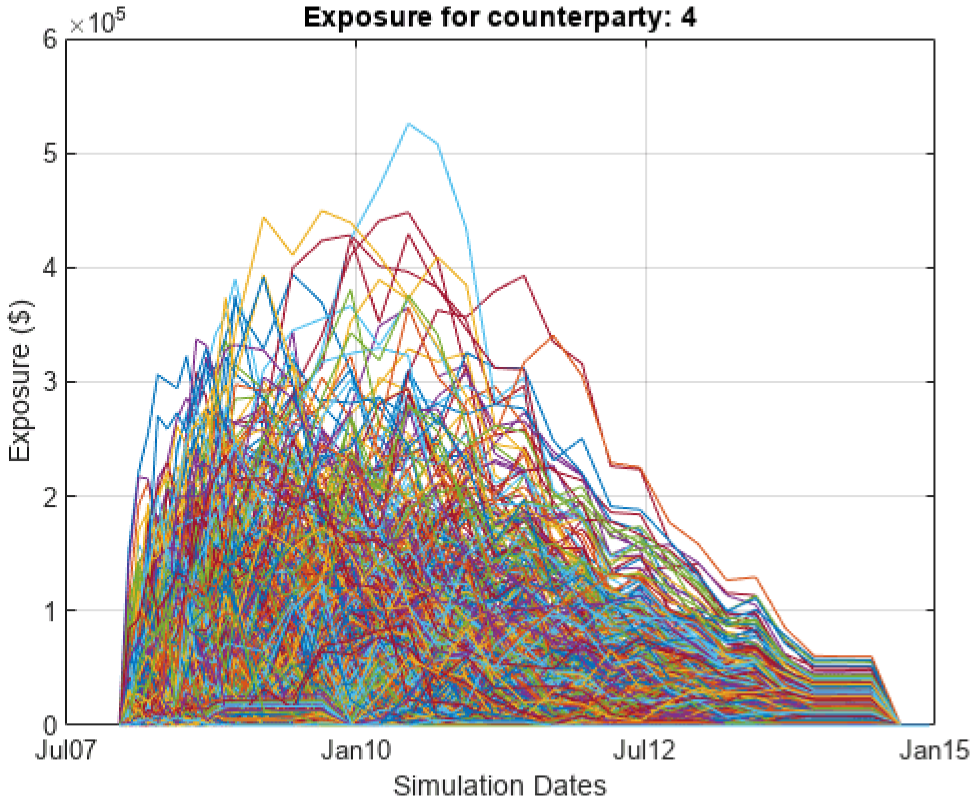


Compute the exposures.

```
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.

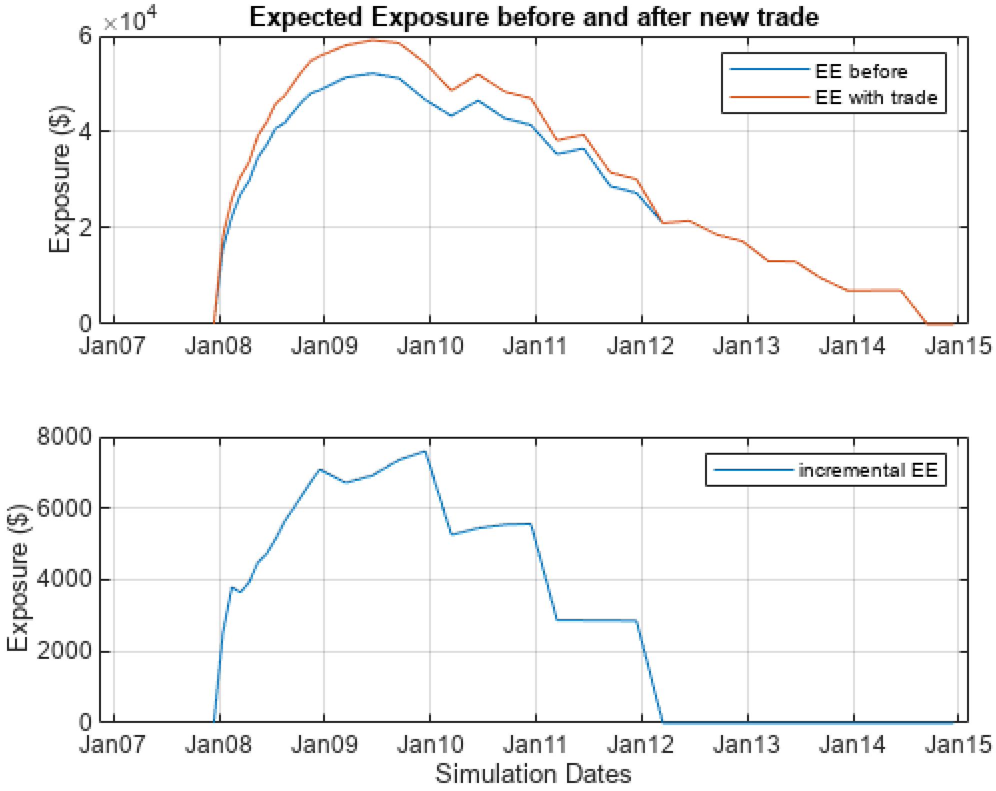
```
newTradeIdx = 3;
newTradeValues = values(:,newTradeIdx,:);

% 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. Use the incremental exposure 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.

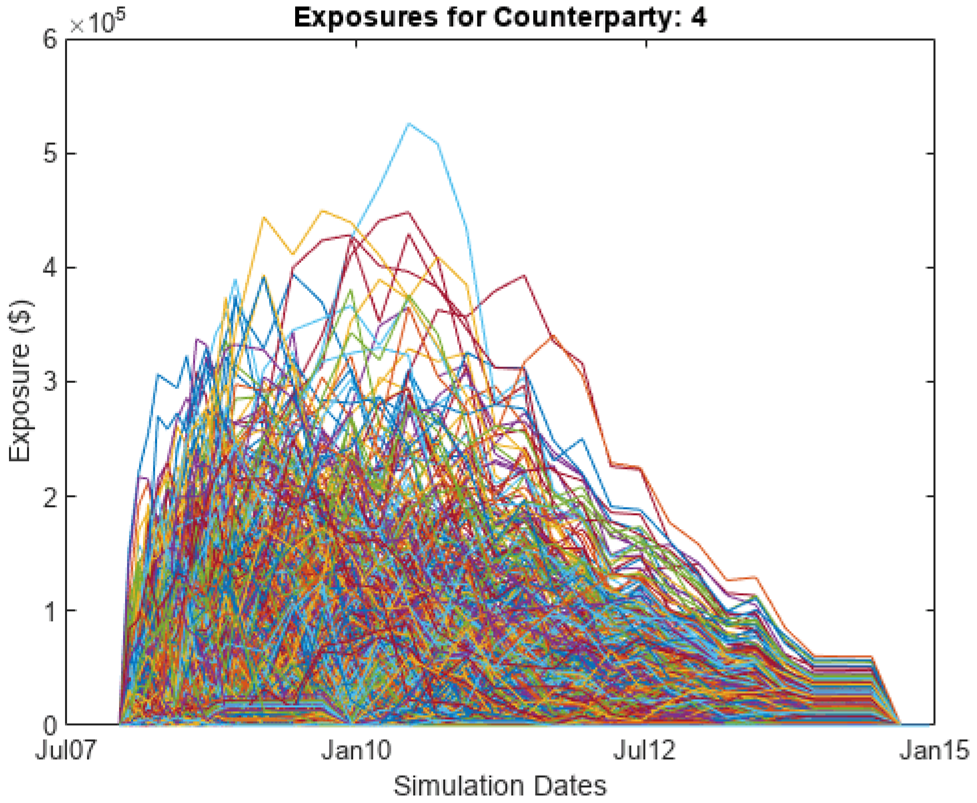
```
cpID = 4;
cpIdx = swaps.Counterparty == cpID;
cpValues = values(:,cpIdx,:);
```

Compute the uncollateralized exposures.

```
exposures = creditexposures(cpValues,swaps.Counterparty(cpIdx),...
'NettingID',swaps.NettingID(cpIdx));
```

View the 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')
```



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=1×4 table
    Counterparty    PeriodOfRisk    Threshold    MinimumTransfer
    ____________    ____________    _________    _______________

         4               14           1e+05           10000
```

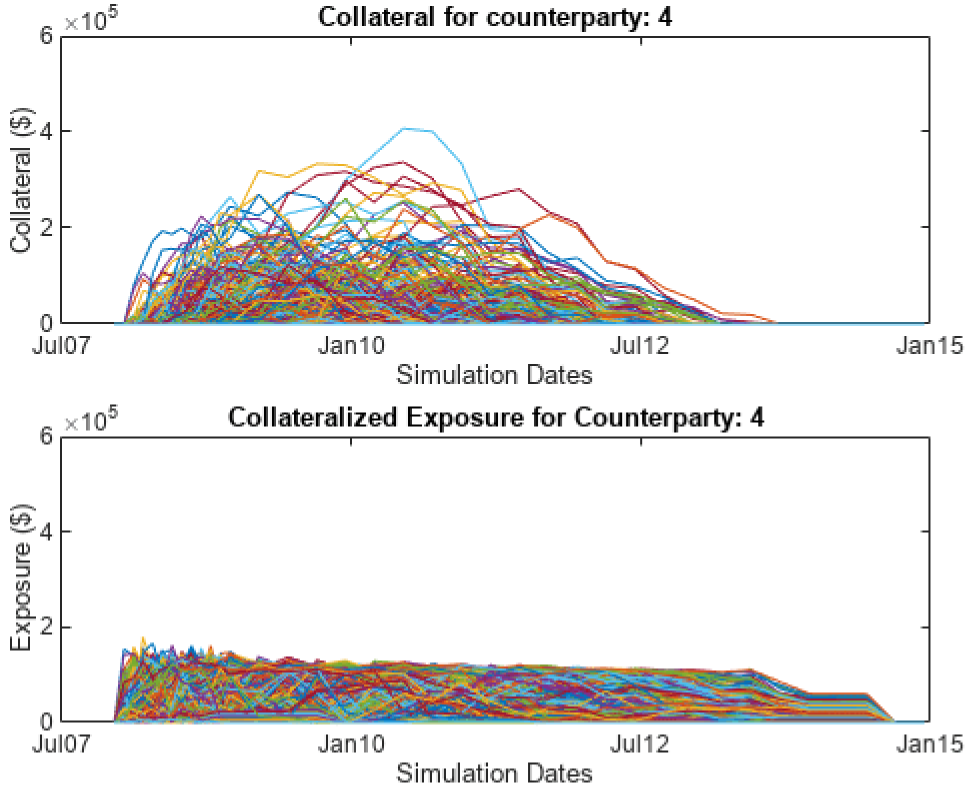
Compute the collateralized exposures.

```
[collatExp, collatcpty, collateral] = creditexposures(cpValues,...
    swaps.Counterparty(cpIdx),'NettingID',swaps.NettingID(cpIdx),...
    'CollateralTable',collateralTable,'Dates',simulationDates);
```

Plot the 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');
```



Collateral for counterparty: 4

Collateralized Exposure for Counterparty: 4

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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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 15-596. If the `CollateralTable` was not specified, this output is empty.

More About

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.

Version History

Introduced in R2014a

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

`exposureprofiles` | `datenum` | `table`

Topics

“Counterparty Credit Risk and CVA” (Financial Instruments Toolbox)
“Wrong Way Risk with Copulas” (Financial Instruments Toolbox)

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.

Load the data (`ccr.mat`) that contains 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.

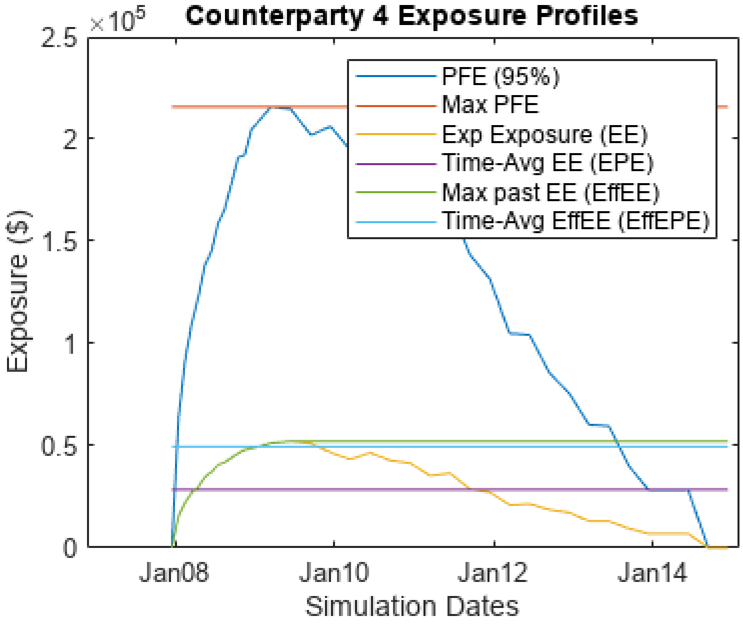
```
 cpProfiles = exposureprofiles(simulationDates,exposures)
```

```
cpProfiles=5×1 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')
```



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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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.

Version History

Introduced in R2014a

References

[1] *Basel II: International Convergence of Capital Measurement and Capital Standards: A Revised Framework - Comprehensive Version.* at `https://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)

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
```

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

datetime array | string array | date character vector

Settlement date of the certificate of deposit, specified as a scalar or a `NCDS`-by-`1` or `1`-by-`NCDS` vector using a datetime array, string array, or date character vectors. The `Settle` date must be before the `Maturity` date.

To support existing code, `cdyield` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Maturity` — Maturity date of certificate of deposit

datetime array | string array | date character vector

Maturity date of the certificate of deposit, specified as a scalar or a `NCDS`-by-`1` or `1`-by-`NCDS` vector using a datetime array, string array, or date character vectors.

To support existing code, `cdyield` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`IssueDate` — Issue date for certificate of deposit

datetime array | string array | date character vector

Issue date for the certificate of deposit, specified as a scalar or a `NCDS`-by-`1` or `1`-by-`NCDS` vector using a datetime array, string array, or date character vectors.

To support existing code, `cdyield` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `string` | `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 2-16.

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.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended
Not recommended starting in R2022b

Although cdyield supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`bndprice` | `cdai` | `cdprice` | `stepcpnprice` | `tbillprice` | `datetime`

Topics

"Coupon Date Calculations" on page 2-20
"Yield Conventions" on page 2-21

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 NaNs 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 = 2×6

   -0.7667    1.5000    1.5000    1.5000    1.5000  101.5000
   -1.8989    2.5000    2.5000    2.5000  102.5000       NaN


CFlowDates = 2×6

      728234      728278      728368      728460      728552      728643
      728234      728278      728460      728643      728825         NaN


TFactors = 2×6

         0    0.2404    0.7403    1.2404    1.7403    2.2404
         0    0.2404    1.2404    2.2404    3.2404       NaN


CFlowFlags = 2×6

     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 datetime 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 datetime array.

```
Settle = datetime(1993,11,1);
Maturity = [datetime(1994,12,15) ; datetime(1995,6,15)];
CouponRate= [0.06; 0.05];
Period = [4; 2];
Basis = [1; 0];
[CFlowAmounts, CFlowDates, TFactors, CFlowFlags] = cfamounts(CouponRate,...
Settle, Maturity, Period, Basis)
```

```
CFlowAmounts = 2×6

   -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
   01-Nov-1993   15-Dec-1993   15-Mar-1994   15-Jun-1994   15-Sep-1994   15-Dec-1994
   01-Nov-1993   15-Dec-1993   15-Jun-1994   15-Dec-1994   15-Jun-1995   NaT


TFactors = 2×6

         0    0.2404    0.7403    1.2404    1.7403    2.2404
         0    0.2404    1.2404    2.2404    3.2404       NaN
```

```
CFlowFlags = 2×6

     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 = 2×8

   -1.2667    1.5000    1.5000    1.5000    1.5000    1.5000    1.5000  101.5000
   -2.3077    2.4932    2.5068    2.4932    2.5000  102.5000       NaN       NaN

CFlowDates = 2×8

      734290      734304      734396      734487      734577      734669      734761      734852
      734290      734304      734487      734669      734852      735035         NaN         NaN

TFactors = 2×8

         0    0.0778    0.5778    1.0778    1.5778    2.0778    2.5778    3.0778
         0    0.0769    1.0769    2.0769    3.0769    4.0769       NaN       NaN

CFlowFlags = 2×8

     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 = 1×10

   -1.8453    2.0000    2.0000    2.0000    2.5000    2.5000    3.0000    3.0000    3.0000  103.0
```

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 = 1×10

   -2.3066    2.5000    2.5000   12.5000    2.2500   12.2500    2.0000    2.0000    2.0000   82.0
```

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 = 2×11

   -1.5217    2.5000    2.5000    2.5000    2.5000    2.5000   12.5000    2.2500   92.2500
   -1.5217    2.5000    2.5000    2.5000    2.5000    2.5000   12.5000    2.2500   12.2500    2.0
```

```
CFDates = 2×11
```

```
      734446      734518      734699      734883      735065      735249      735430      735614
      734446      734518      734699      734883      735065      735249      735430      735614


TFactors = 2×11

         0    0.3913    1.3913    2.3913    3.3913    4.3913    5.3913    6.3913    7.3913
         0    0.3913    1.3913    2.3913    3.3913    4.3913    5.3913    6.3913    7.3913    8.


CFFlags = 2×11

     0     3     3     3     3     3    13     3     4   NaN   NaN
     0     3     3     3     3     3    13     3    13     3     4


CFPrincipal = 2×11

     0     0     0     0     0     0    10     0    90   NaN   NaN
     0     0     0     0     0     0    10     0    10     0    80
```

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. This means that the corresponding `CouponRate` and `Face` value applies "on or before" the specified ending date.

Data Types: `double` | `cell` | `char`

`Settle` — Settlement date of bond
datetime array | string array | date character vector

Settlement date of the bond, specified as a scalar or a `NBONDS`-by-`1` vector using a datetime array, string array, or date character vectors. The `Settle` date must be before the `Maturity` date.

To support existing code, `cfamounts` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Maturity` — Maturity date of bond
datetime array | string array | date character vector

Maturity date of the bond, specified as a scalar or a NBONDS-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, cfamounts also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: [CFlowAmounts, CFlowDates, TFactors, CFlowFlags] = ... cfamounts(CouponRate,Settle, Maturity,'Period',4,'Basis',3,'AdjustCashFlowsBasis',true,'BusinessDayConvention','modifiedfollow')

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 the comma-separated pair consisting of 'Period' and a 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 the comma-separated pair consisting of 'Basis' and a 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 2-16.

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 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
datetime array | string array | date character vector

Bond issue date (the date the bond begins to accrue interest), specified as the comma-separated pair consisting of `'IssueDate'` and a scalar or a `NBONDS`-by-`1` vector using a datetime array, string array, or date character vectors. 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.

To support existing code, `cfamounts` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`FirstCouponDate` — Irregular or normal first coupon date
datetime array | string array | date character vector

Irregular or normal first coupon date, specified as the comma-separated pair consisting of `'FirstCouponDate'` and a scalar or a `NBONDS`-by-`1` vector using a datetime array, string array, or date character vectors.

If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

Note When `FirstCouponDate` and `LastCouponDate`

are both specified, the `FirstCouponDate` takes precedence in determining the coupon payment structure. If `FirstCouponDate` is not specified, then `LastCouponDate` determines the coupon structure of the bond.

To support existing code, `cfamounts` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`LastCouponDate` — Irregular or normal last coupon date
datetime array | string array | date character vector

Irregular or normal last coupon date, specified as the comma-separated pair consisting of `'LastCouponDate'` and a scalar or a `NBONDS`-by-`1` vector using a datetime array, string array, or date character vectors.

Note When `FirstCouponDate` and `LastCouponDate` are both specified, the `FirstCouponDate` takes precedence in determining the coupon payment structure. If `FirstCouponDate` is not specified, then `LastCouponDate` determines the coupon structure of the bond.

To support existing code, `cfamounts` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`StartDate` — Forward starting date of coupon payments

datetime array | string array | date character vector

Forward starting date of coupon payments after the `Settle` date, specified as the comma-separated pair consisting of `'StartDate'` and a scalar or a `NBONDS`-by-`1` vector using a datetime array, string array, or date character vectors.

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.

To support existing code, `cfamounts` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Face` — Face value of bond

`100` (default) | numeric

Face value of the bond, specified as the comma-separated pair consisting of `'Face'` and 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. This means that the corresponding `CouponRate` and `Face` value applies "on or before" the specified ending date.

When the corresponding `Face` value is used to compute the coupon cashflow on the specified ending date. Three things happen on the specified ending date:

1. The last coupon corresponding to the current `Face` value is paid.
2. The principal differential (between the current and the next `Face` value) is paid.
3. The date marks the beginning of the period with the next `Face` value, for which the cashflow does not occur until later.

Data Types: double | cell | char

AdjustCashFlowsBasis — Adjusts cash flows according to accrual amount based on actual period day count
false (default) | logical with a value of true or false

Adjusts cash flows according to the accrual amount based on the actual period day count, specified as the comma-separated pair consisting of 'AdjustCashFlowsBasis' and 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 the comma-separated pair consisting of 'BusinessDayConvention' and 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 uses2, ICMA uses 1 (default) | integer with value of 1, 2, 3, 4, 6, or 12

Compounding frequency for yield calculation, specified as the comma-separated pair consisting of 'CompoundingFrequency' and 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 the comma-separated pair consisting of 'DiscountBasis' and 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 2-16.

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) | datetime array | string array | date character vector

Dates for holidays, specified as the comma-separated pair consisting of 'Holidays' and a NHOLIDAYS-by-1 vector using a datetime array, string array, or date character vectors. Holidays are used in computing business days.

To support existing code, cfamounts also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | 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, specified as the comma-separated pair consisting of 'PrincipalType' and a value of '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 strings s or date character vectors, then CFlowDates is returned as a serial date number. Use the function datestr to convert serial date numbers to formatted date character vectors. 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 15-617.

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.

More About

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.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although `cfamounts` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)

y =

        2021
```

There are no plans to remove support for serial date number inputs.

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` | `cpncount` | `cftimes` | `cpndaten` | `cpndatenq` | `cpndatep` | `cpndaysn` | `cpndaysp` | `datetime` | `cpndatepq`

Topics

"Analyzing and Computing Cash Flows" on page 2-11

External Websites

Asset Liability Management Using MATLAB (3 min 58 sec)

cfconv

Cash flow convexity

Syntax

```
CFlowConvexity = cfconv(CashFlow,Yield)
```

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
```

Input Arguments

`CashFlow` — Cash flow
vector of real numbers

Cash flow, specified as a vector of real numbers.

Data Types: `double`

`Yield` — Periodic yield
scalar decimal

Periodic yield, specified as a scalar decimal.

Data Types: `double`

Output Arguments

`CFlowConvexity` — Convexity
scalar

Convexity returned as a scalar.

Version History

Introduced before R2006a

See Also

`bndconvp` | `bndconvy` | `bnddurp` | `bnddury` | `cfdur`

Topics

"Analyzing and Computing Cash Flows" on page 2-11

cfdates

Cash flow dates for fixed-income security

Syntax

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

Description

CFlowDates = cfdates(Settle,Maturity) generates a matrix of actual cash flow payment dates for NUMBONDS fixed income securities. All cash flow dates are determined regardless of whether the first and last coupon periods are normal, long or short.

CFlowDates = cfdates(___ ,Period,Basis,EndMonthRule,IssueDate,FirstCouponDate, LastCouponDate) specifies options using one or more optional arguments in addition to the input arguments in the previous syntax.

Examples

Obtain Cash Flow Dates for Fixed-Income Security

Compute the cash flow dates given the Settle and Maturity dates.

```
CFlowDates = cfdates('14 Mar 1997', '30 Nov 1998', 2, 0, 1)
```

```
CFlowDates = 1×4

      729541      729724      729906      730089
```

```
datestr(CFlowDates)
```

```
ans = 4x11 char array
    '31-May-1997'
    '30-Nov-1997'
    '31-May-1998'
    '30-Nov-1998'
```

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:

```
CFlowDates = cfdates('14-Mar-1997', datetime('30-Nov-1998','Locale','en_US'), 2, 0, 1)
```

```
CFlowDates = 1x4 datetime
   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 = 3×4

      729480      729663         NaN         NaN
      729510      729694      729875      730059
      729541      729724      729906      730089
```

To look at the cash-flow dates for the last security:

```
datestr(CFlowDates(3,:))
```

```
ans = 4x11 char array
    '31-May-1997'
    '30-Nov-1997'
    '31-May-1998'
    '30-Nov-1998'
```

Input Arguments

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors. `Settle` must be earlier than `Maturity`.

To support existing code, `cfdates` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `cfdates` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Period` — Coupons per year of the bond

2 (semiannual) (default) | vector of positive integers from the set `[1,2,3,4,6,12]`

(Optional) Coupons per year of the bond, specified as a vector of positive integers from the set `[1,2,3,4,6,12]`.

Data Types: `double`

`Basis` — Day-count basis

`0` (actual/actual) (default) | positive integers of the set `[1...13]` | vector of positive integers of the set `[1...13]`

(Optional) Day-count basis, specified as positive integers 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 2-16.

Data Types: double

EndMonthRule — End-of-month rule flag
1 (in effect) (default) | nonnegative integer 0 or 1

(Optional) 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
cash flow payment dates are determined from other inputs (default) | datetime array | string array | date character vector

(Optional) Bond Issue date, specified as a scalar or a NUMBONDS-by-1 or 1-by-NUMBONDS vector using a datetime array, string array, or date character vectors.

If you do not specify an IssueDate, the cash flow payment dates are determined from other inputs.

To support existing code, cfdates also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

FirstCouponDate — Irregular or normal first coupon date
cash flow payment dates are determined from other inputs (default) | datetime array | string array | date character vector

Irregular or normal first coupon date, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using a datetime array, string array, or date character vectors.

If you do not specify a `FirstCouponDate`, the cash flow payment dates are determined from other inputs.

To support existing code, `cfdates` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`LastCouponDate` — Irregular or normal last coupon date
cash flow payment dates are determined from other inputs (default) | datetime array | string array | date character vector

Irregular or normal last coupon date, specified as a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using a datetime array, string array, or date character vectors.

If you do not specify a `LastCouponDate`, the cash flow payment dates are determined from other inputs.

To support existing code, `cfdates` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

Output Arguments

`CFlowDates` — Actual cash flow payment dates
matrix

Actual cash flow payment dates, returned as a `N`-row matrix of dates in datetime format (if inputs are in datetime format). `CFlowDates` 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.

If all of the inputs for `Settle`, `Maturity`, `IssueDate`, `FirstCouponDate`, or `LastCouponDate` are either strings or date character vectors, then `CFlowDates` is returned as a serial date number. Use the function `datestr` to convert serial date numbers to formatted date character vectors.

If any of the inputs for `Settle`, `Maturity`, `IssueDate`, `FirstCouponDate`, or `LastCouponDate` 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`.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended
Not recommended starting in R2022b

Although cfdates supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

accrfrac | cfamounts | cftimes | cpncount | cpndaten | cpndatenq | cpndatep | cpndaysn | cpndaysp | cpnpersz | datetime | cpndatepq

Topics

“Analyzing and Computing Cash Flows” on page 2-11

cfdatesq

Quasi-coupon dates for fixed-income security

Syntax

```
QuasiCouponDates = cfdatesq(Settle,Maturity)
QuasiCouponDates = cfdatesq(___,Period,Basis,EndMonthRule,IssueDate,
FirstCouponDate,LastCouponDate,PeriodsBeforeSettle,PeriodsAfterMaturity)
```

Description

`QuasiCouponDates = cfdatesq(Settle,Maturity)` returns a matrix of quasi-coupon dates expressed in 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. `NaN`s 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.

`QuasiCouponDates = cfdatesq(___,Period,Basis,EndMonthRule,IssueDate, FirstCouponDate,LastCouponDate,PeriodsBeforeSettle,PeriodsAfterMaturity)` specifies options using one or more optional arguments in addition to the input arguments in the previous syntax.

Examples

Obtain Quasi-Coupon Dates for Fixed-Income Security

Compute the quasi-coupon dates given the `Settle` and `Maturity` dates.

```
QuasiCouponDates = cfdatesq('14-Mar-1997', '30-Nov-1998', 2, 0, 1)
```

```
QuasiCouponDates = 1×4

      729541      729724      729906      730089
```

If any of the inputs for `Settle`, `Maturity`, `IssueDate`, `FirstCouponDate`, or `LastCouponDate` are datetime arrays, then `QuasiCouponDates` is returned as a datetime array. For example:

```
QuasiCouponDates = cfdatesq('14-Mar-1997', datetime('30-Nov-1998','Locale','en_US'), 2, 0, 1)
QuasiCouponDates = 1x4 datetime
   31-May-1997   30-Nov-1997   31-May-1998   30-Nov-1998
```

Input Arguments

Settle — Settlement date

datetime array | string array | date character vector

Settlement date, specified as an NINST-by-1 vector using a datetime array, string array, or date character vectors. Settle must be earlier than Maturity.

To support existing code, cfdatesq also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

Maturity — Maturity date

datetime array | string array | date character vector

Maturity date, specified as an NINST-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, cfdatesq also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

Period — Coupons per year of the bond

2 (semiannual) (default) | vector of positive integers from the set [1,2,3,4,6,12]

(Optional) Coupons per year of the bond, specified as a vector of positive integers from the set [1,2,3,4,6,12].

Data Types: double

Basis — Day-count basis

0 (actual/actual) (default) | positive integers of the set [1...13] | vector of positive integers of the set [1...13]

(Optional) Day-count basis, specified as positive integers 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 2-16.

Data Types: double

EndMonthRule — End-of-month rule flag
1 (in effect) (default) | nonnegative integer 0 or 1

(Optional) 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
cash flow payment dates are determined from other inputs (default) | datetime array | string array | date character vector

(Optional) Bond Issue date, specified as a scalar or a NUMBONDS-by-1 or 1-by-NUMBONDS vector using a datetime array, string array, or date character vectors.

If you do not specify an IssueDate, the cash flow payment dates are determined from other inputs.

To support existing code, cfdatesq also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

FirstCouponDate — Irregular or normal first coupon date
cash flow payment dates are determined from other inputs (default) | datetime array | string array | date character vector

Irregular or normal first coupon date, specified as a scalar or a NUMBONDS-by-1 or 1-by-NUMBONDS vector using a datetime array, string array, or date character vectors.

If you do not specify a FirstCouponDate, the cash flow payment dates are determined from other inputs.

To support existing code, cfdatesq also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

LastCouponDate — Irregular or normal last coupon date
cash flow payment dates are determined from other inputs (default) | datetime array | string array | date character vector

Irregular or normal last coupon date, specified as a scalar or a NUMBONDS-by-1 or 1-by-NUMBONDS vector using a datetime array, string array, or date character vectors.

If you do not specify a LastCouponDate, the cash flow payment dates are determined from other inputs.

To support existing code, cfdatesq also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

PeriodsBeforeSettle — Number of quasi-coupon dates on or before settlement to include
0 (default) | nonnegative integer

(Optional) Number of quasi-coupon dates on or before settlement to include, specified as a nonnegative integer.

Data Types: double

PeriodsAfterMaturity — Number of quasi-coupon dates after maturity to include
0 (default) | nonnegative integer

(Optional) Number of quasi-coupon dates after maturity to include, specified as a nonnegative integer.

Data Types: double

Output Arguments

QuasiCouponDates — Quasi-coupon dates
matrix

Quasi-coupon dates, returned as a N-row matrix of dates in datetime format (if any inputs are in datetime format). 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 all of the inputs for Settle, Maturity, IssueDate, FirstCouponDate, or LastCouponDate are either strings or date character vectors, then QuasiCouponDates is returned as a serial date number. Use the function datestr to convert serial date numbers to formatted date character vectors.

If any of the inputs for Settle, Maturity, IssueDate, FirstCouponDate, or LastCouponDate are datetime arrays, then QuasiCouponDates is returned as a datetime array.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although cfdatesq supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

accrfrac | cfamounts | cftimes | cpncount | cpndaten | cpndatenq | cpndatep | cpndaysn | cpndaysp | cpnpersz | datetime | cpndatepq

Topics

“Analyzing and Computing Cash Flows” on page 2-11

cfdur

Cash-flow duration and modified duration

Syntax

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

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
```

Input Arguments

`CashFlow` — Cash flow
vector of real numbers | matrix of real numbers

Cash flow, specified as a vector or matrix of real numbers. When using a matrix, each column of the matrix is a separate `CashFlow`.

Data Types: `double`

`Yield` — Periodic yield
scalar decimal | vector of decimals

Periodic yield, specified as a scalar decimal or a vector of decimals.

Data Types: `double`

Output Arguments

`Duration` — Duration
vector

Duration returned as a scalar or vector.

`ModDuration` — Modified duration
vector

Modified duration, returned as a scalar or vector.

Version History

Introduced before R2006a

See Also

`bndconvp` | `bndconvy` | `bnddurp` | `bnddury` | `cfconv`

Topics

“Analyzing and Computing Cash Flows” on page 2-11

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

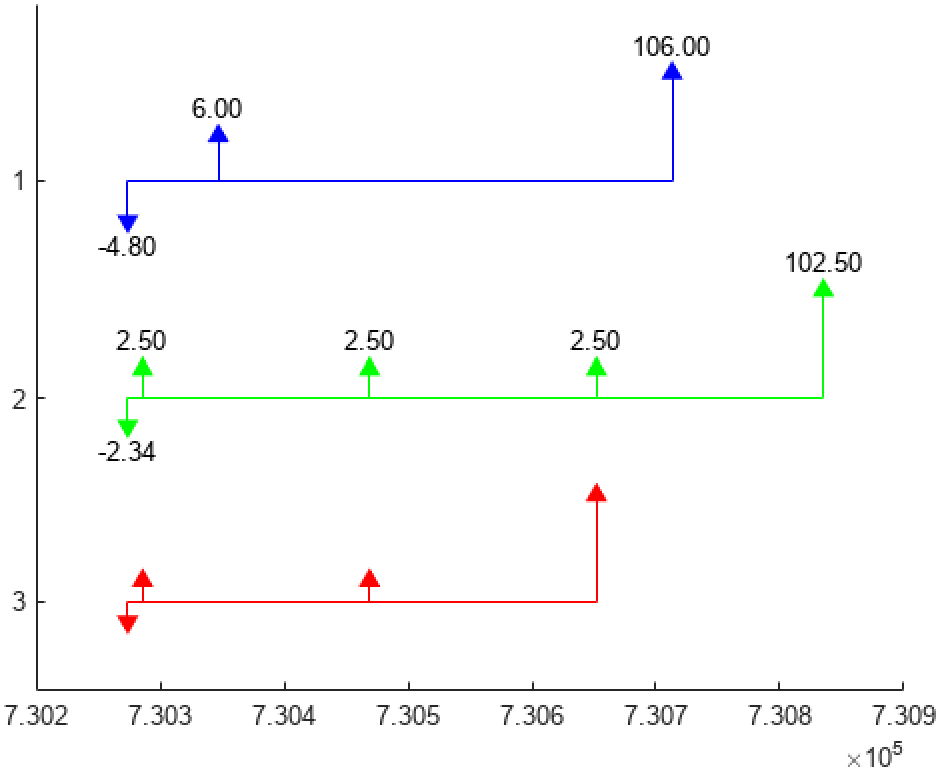
Define `CFlowAmounts` and `CFlowDates` using the `cfamounts` function.

```
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 = 3×5

   -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 = 3×5

      730274      730347      730713         NaN         NaN
      730274      730286      730469      730652      730835
      730274        730286        730469        730652           NaN
```

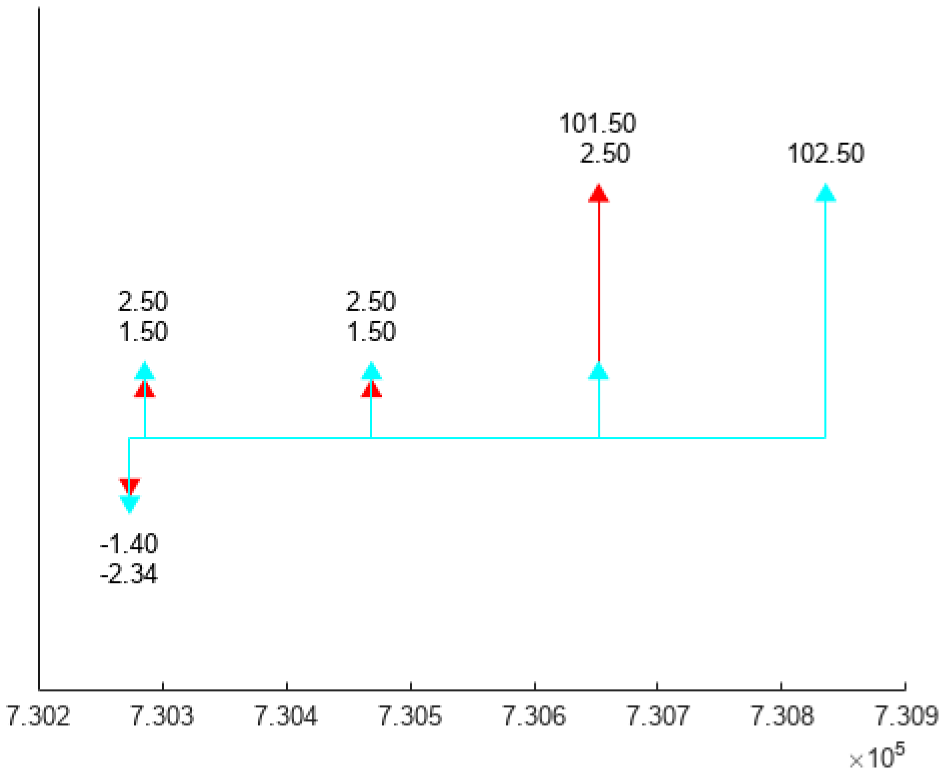
Plot all cash flows on the same axes, and label the first two.

```
cfplot(CFlowDates, CFlowAmounts, 'ShowAmnt', [1 2])
```



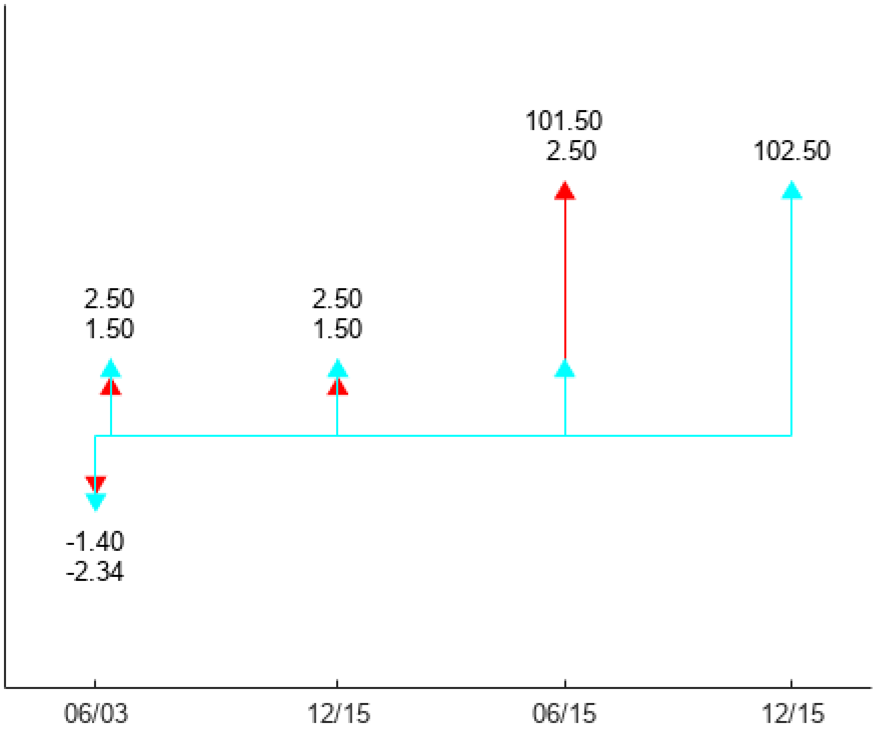
Group the second and third cash flows.

```
figure;
cfplot(CFlowDates, CFlowAmounts, 'Groups', {[2 3]}, 'ShowAmnt', 1);
```



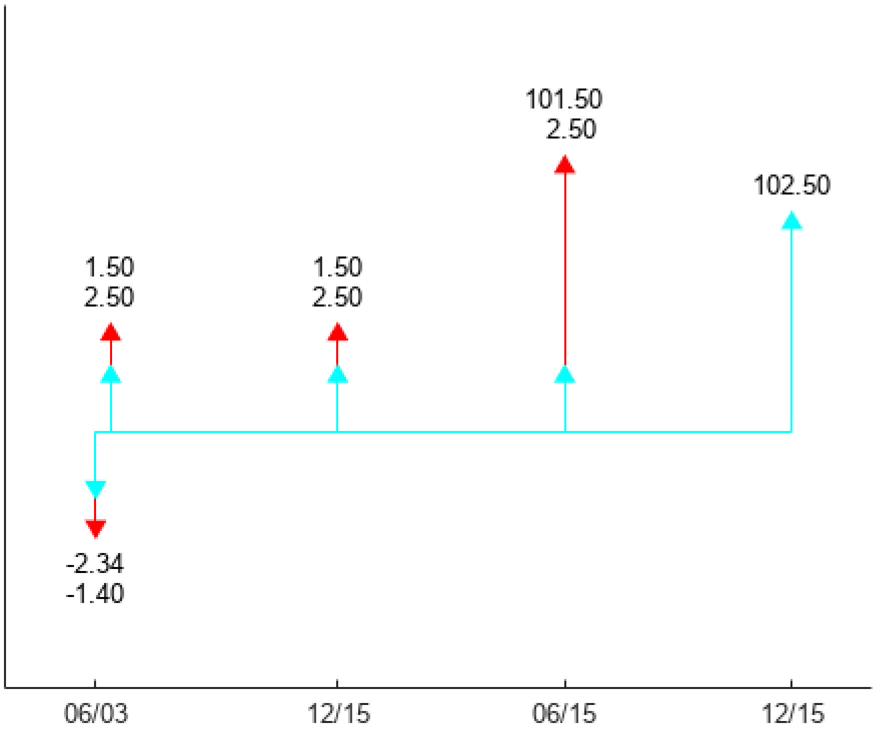
Format the date axis and place ticks on actual cash flow dates.

```
figure;
cfplot(CFlowDates, CFlowAmounts, 'Groups', {[2 3]}, 'ShowAmnt', 1, ...
'DateFormat', 6, 'DateSpacing', 100);
```



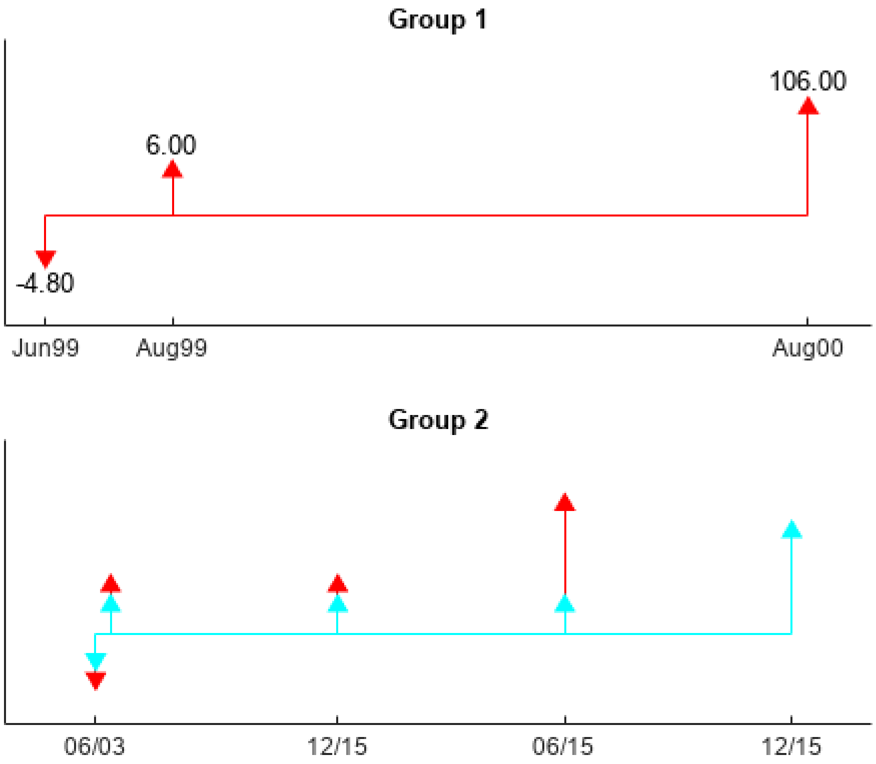
Stack the cash flow arrows occurring on the same dates.

```
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.

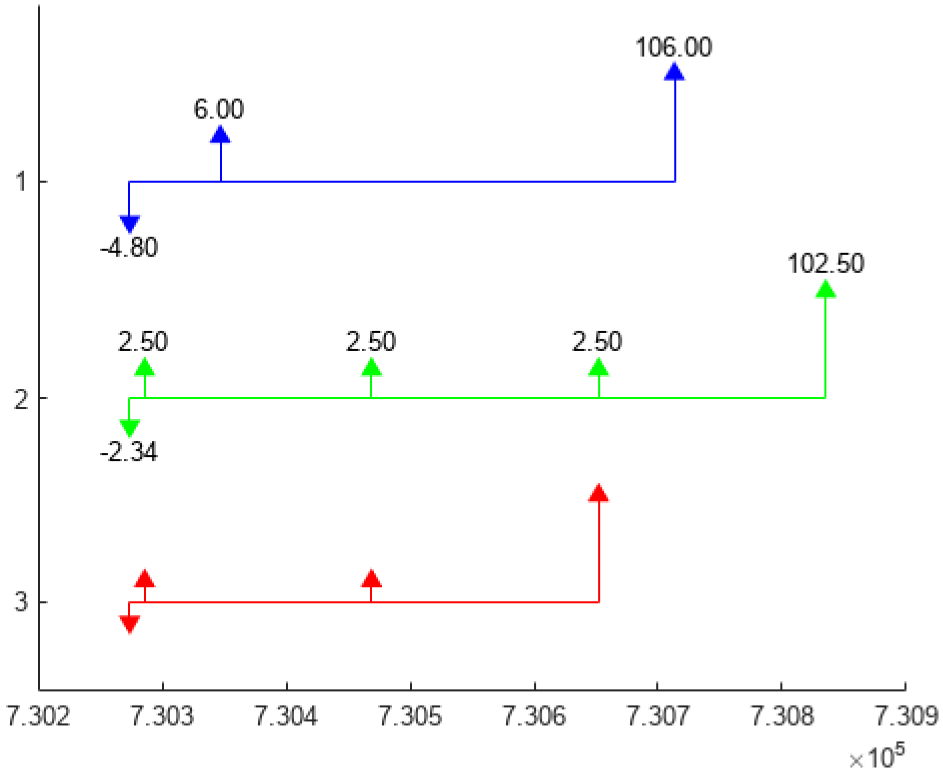
```
figure;
[h, axes_handle] = cfplot(CFlowDates, CFlowAmounts, ...
'Groups', {[1] [2 3]}, 'ShowAmnt', 1, 'Stacked', 2, ...
'DateSpacing', [1 60 2 100], 'DateFormat', [1 12 2 6]);
title(axes_handle(1), 'Group 1', 'FontWeight', 'bold');
title(axes_handle(2), 'Group 2', 'FontWeight', 'bold');
```



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, 'ShowAmnt',
```



Plot Cash Flows for Swap

Define the swap using the swapbyzero function.

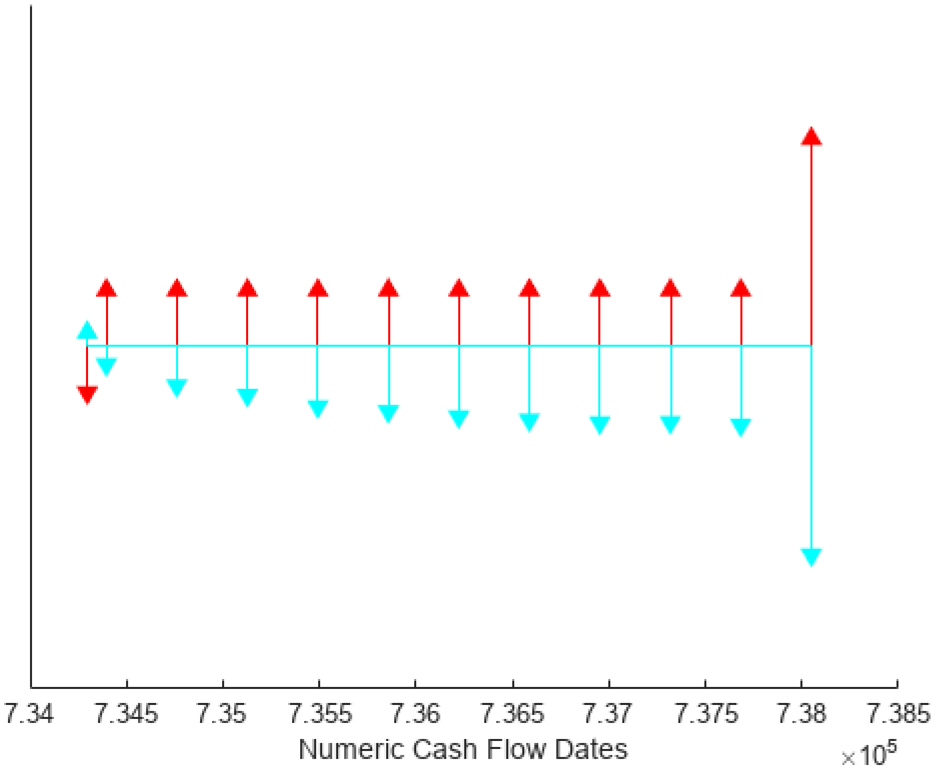
```
Settle = datetime(2010,6,8);
RateSpec = intenvset('Rates', [.005 .0075 .01 .014 .02 .025 .03]',...
'StartDates',Settle, 'EndDates',[datetime(2010,12,8),datetime(2011,6,8),datetime(2012,6,8),datet
Maturity = datenum('15-Sep-2020');
LegRate = [.025 50];
LegType = [1 0]; % fixed/floating
LatestFloatingRate = .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 = 1×12

   -1.8219    2.5000    2.5000    2.5000    2.5000    2.5000    2.5000    2.5000    2.5000    2.5
```

```
RecCFDates = 1×12

      734297       734396       734761       735127       735492       735857       736222       736588

PayCF = 1×12

   -0.3644     0.5000     1.4048     1.9823     2.8436     3.2842     3.8218     4.1733     4.5164     4.4

PayCFDates = 1×12

      734297       734396       734761       735127       735492       735857       736222       736588
```

Define `CFlowDates` and `CFlowAmounts` for the swap and generate a cash flow plot using `cfplot`.

```
CFlowDates = [PayCFDates;RecCFDates];
CFlowAmounts = [-PayCF;RecCF];
cfplot(CFlowDates, CFlowAmounts, 'Groups', {[1 2]});
xlabel('Numeric Cash Flow Dates');
```



Input Arguments

`CFlowDates` — Matrix of dates for cash flows

vector

Matrix of datetime arrays for cash flows, specified as a NINST-by-(Number of cash flows) matrix of cash flow dates using datetime format, with empty entries padded with NaNs.

Each row of the CFlowDates matrix represents an instrument so that CFlowDates(k,:) is the vector of cash flow dates for the kth instrument. Rows are padded with trailing NaNs if the number of cash flows is not the same for all instruments.

cfamounts can be used to generate CFlowDates.

Data Types: datetime

CFlowAmounts — Matrix of cash flow amounts
vector

Matrix of cash flow amounts, specified as a NINST-by-(Number of cash flows) matrix of cash flow amounts, with empty entries padded with NaNs. The CFlowAmounts matrix must be the same size as CFlowDates.

cfamounts can be used to generate CFlowAmounts.

Data Types: double

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

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 as the comma-separated pair consisting of 'Groups' and 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 as the comma-separated pair consisting of 'Stacked' and 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 as the comma-separated pair consisting of 'ShowAmnt' and 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 as the comma-separated pair consisting of 'DateSpacing' and 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, specified as the comma-separated pair consisting of 'DateFormat' and 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

DateFormNum	Example
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.

Version History

Introduced in R2013a

See Also

cfamounts | cfdates | swapbyzero | datetime

Topics

“Analyzing and Computing Cash Flows” on page 2-11

cfport

Portfolio form of cash flow amounts

Syntax

```
[CFBondDate,AllDates,AllTF,IndByBond] = cfport(CFlowAmounts,CFlowDates)
[CFBondDate,AllDates,AllTF,IndByBond] = cfport( ___ ,TFactors)
```

Description

`[CFBondDate,AllDates,AllTF,IndByBond] = cfport(CFlowAmounts,CFlowDates)` 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.

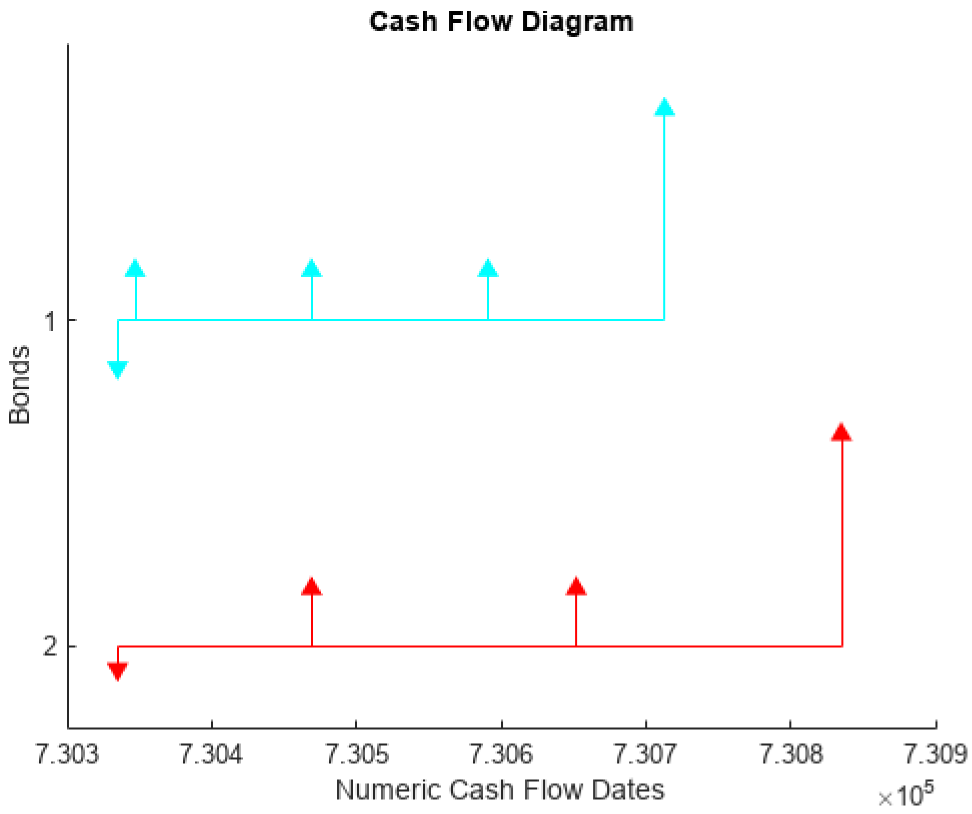
`[CFBondDate,AllDates,AllTF,IndByBond] = cfport( ___ ,TFactors)` specifies options using one or more optional arguments in addition to the input arguments in the previous syntax.

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 = 2×7

   -1.8000    2.0000    2.0000    2.0000         0  102.0000         0
   -0.6694         0    2.5000         0    2.5000         0  102.5000


AllDates = 7×1

      730335
      730347
      730469
      730591
      730652
      730713
      730835
```

```
AllTF = 7×1

         0
    0.0663
    0.7322
    1.3989
    1.7322
    2.0663
    2.7322


IndByBond = 2×5

     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(1999,8,3);
Maturity = [datetime(2000,8,15) ; datetime(2000,12,15)];
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 = 2×7

   -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
   03-Aug-1999
   15-Aug-1999
   15-Dec-1999
   15-Apr-2000
   15-Jun-2000
   15-Aug-2000
   15-Dec-2000


AllTF = 7×1
```

```
         0
    0.0663
    0.7322
    1.3989
    1.7322
    2.0663
    2.7322


IndByBond = 2×5

     1     2     3     4     6
     1     3     5     7   NaN
```

Input Arguments

CFlowAmounts — Cash flow amounts
matrix

Cash flow amounts, specified as number of bonds (NUMBONDS) by number of cash flows (NUMCFS) matrix with entries listing cash flow amounts corresponding to each date in CFlowDates.

Data Types: double

CFlowDates — Cash flow dates
datetime array | string array | date character vector

Cash flow dates, specified as an NUMBONDS-by-NUMCFS matrix with rows listing cash flow dates using a datetime array, string array, or date character vectors for each bond and padded with NaNs. 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.

To support existing code, cfport also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

TFactors — Time between settlement and the cash flow date
matrix

(Optional) Time between settlement and the cash flow date, specified as an NUMBONDS-by-NUMCFS matrix with entries listing the time between settlement and the cash flow date measured in semiannual coupon periods.

Data Types: double

Output Arguments

CFBondDate — Cash flows indexed by bond and by date
matrix

Cash flows indexed by bond and by date, returned as an NUMBONDS by number of dates (NUMDATES) matrix. 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 — List of all dates that have any cash flow from the bond portfolio
matrix

List of all dates that have any cash flow from the bond portfolio, returned as an NUMDATES-by-1 matrix. The AllDates matrix is expressed in datetime format (if CFlowDates is in datetime format).

AllTF — Time factors corresponding to the dates in AllDates
matrix

Time factors corresponding to the dates in AllDates, returned as an NUMDATES-by-1 matrix. If TFactors is not entered, AllTF contains the number of days from the first date in AllDates.

IndByBond — Indices by bond
matrix

Indices by bond, returned as an NUMBONDS-by-NUMCFS matrix. The *i*th row contains a list of indices into AllDates where the *i*th bond has cash flows. Since some bonds have more cash flows than others, the matrix is padded with NaNs.

Version History
Introduced before R2006a

R2022b: Serial date numbers not recommended
Not recommended starting in R2022b

Although cfport supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)

y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also
cfamounts | cfplot | datetime

Topics
“Analyzing and Computing Cash Flows” on page 2-11

cfprice

Compute price for cash flow given yield to maturity

Syntax

```
Price = cfprice(CFlowAmounts,CFlowDates,Yield,Settle)
Price = cfprice( ___ ,Name,Value)
```

Description

`Price = cfprice(CFlowAmounts,CFlowDates,Yield,Settle)` computes a price given yield for a cash flow.

`Price = cfprice( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

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 = datetime(2003,7,1);
Yield = .05;
CFAmounts = [30;40;30];
CFDates = [datetime(2004,7,15) ; datetime(2005,7,15) ; datetime(2006,7,15)];
```

Compute the `Price`.

```
Price = cfprice(CFAmounts, CFDates, Yield, Settle)
```

```
Price = 3×1

   28.4999
   36.1689
   25.8195
```

Input Arguments

`CFlowAmounts` — Cash flow amounts

vector

Cash flow amounts, specified as an `NINST`-by-`MOSTCFS` matrix. 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.

Data Types: `double`

`CFlowDates` — Cash flow dates

datetime array | string array | date character vector

Cash flow dates, specified as an `NINST`-by-`MOSTCFS` matrix using a datetime array, string array, or date character vectors. Each entry contains the date of the corresponding cash flow in `CFlowAmounts`.

To support existing code, `cfprice` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Yield` — Yields

vector

Yields specified as an `NINST`-by-`1` vector.

Data Types: `double`

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date, specified as an `NMBS`-by-`1` vector using a datetime array, string array, or date character vectors. The `Settle` date is the date on which the cash flows are priced.

To support existing code, `cfprice` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Note An optional input of size `NINST`-by-`1` is also acceptable as a single value applicable to all contracts. Single values are internally expanded to an array of size `NINST`-by-`1`.

Example: `Price = cfprice(CFlowAmounts,CFlowDates,Yield,Settle,'Basis',4,'CompoundingFrequency',4)`

`Basis` — Day-count basis

`0` (actual/actual) (default) | positive integers of the set `[1...13]` | vector of positive integers of the set `[1...13]`

Day-count basis, specified as the comma-separated pair consisting of `'Basis'` and a positive integer using a `N`-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 2-16.

Data Types: `double`

`CompoundingFrequency` — Compounding frequency
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 the comma-separated pair consisting of `'CompoundingFrequency'` and 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`

Output Arguments

`Price` — Price of cash flows
vector

Price of cash flows, returned as an `NINST`-by-`1` vector.

Version History

Introduced in R2012a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although `cfprice` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`cfbyzero` | `cfyield` | `cfspread` | `datetime`

Topics

"Analyzing and Computing Cash Flows" on page 2-11

cfspread

Compute spread over yield curve for cash flow

Syntax

```
Spread = cfspread(RateSpec,Price,CFlowAmounts,CFlowDates,Settle)
Spread = cfspread( ___ ,Name,Value)
```

Description

`Spread = cfspread(RateSpec,Price,CFlowAmounts,CFlowDates,Settle)` computes spread over a yield curve for a cash flow.

`Spread = cfspread( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

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 = datetime(2003,7,1) %datenum('01-Jul-2003');
```

```
Settle = datetime
   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 = [datetime(2004,7,15); datetime(2005,7,15) ; datetime(2006,7,15)];

Spread = cfspread(RateSpec, Price, CFAmounts, CFDates, Settle)
```

```
Spread = 3×1
10^3 ×

   -8.7956
   -4.0774
   -3.7073
```

Input Arguments

RateSpec — Interest-rate specification for initial risk-free rate curve

structure

Interest-rate specification for initial risk-free rate curve, specified by the RateSpec obtained from intenvset. For information on the interest-rate specification, see intenvset.

Data Types: struct

Price — Price of cash flows

vector

Price of cash flows, specified as an NINST-by-1 vector.

Data Types: double

CFlowAmounts — Cash flow amounts

vector

Cash flow amounts, specified as an NINST-by-MOSTCFS matrix . Each row is a list of cash flow values for one instrument. If an instrument has fewer than MOSTCFS cash flows, the end of the row is padded with NaNs.

Data Types: double

CFlowDates — Cash flow dates

datetime array | string array | date character vector

Cash flow dates, specified as an NINST-by-MOSTCFS matrix using a datetime array, string array, or date character vectors. Each entry contains the date of the corresponding cash flow in CFlowAmounts.

To support existing code, cfspread also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

Settle — Settlement date

datetime array | string array | date character vector

Settlement date, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors. The `Settle` date is the date on which the cash flows are priced.

To support existing code, `cfspread` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Note An optional input of size `NINST`-by-`1` is also acceptable as a single value applicable to all contracts. Single values are internally expanded to an array of size `NINST`-by-`1`.

Example: `Spread = cfspread(RateSpec,Price,CFlowAmounts,CFlowDates,Settle,'Basis',4)`

`Basis` — Day-count basis

`0` (actual/actual) (default) | positive integers of the set `[1...13]` | vector of positive integers of the set `[1...13]`

Day-count basis, specified as the comma-separated pair consisting of `'Basis'` and 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 2-16.

Data Types: `double`

Output Arguments

Spread — Spread of cash flows over a zero curve

vector

Spread of cash flows over a zero curve, returned as an `NINST`-by-`1` vector. The `Spread` is expressed in basis points.

Version History

Introduced in R2012a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although `cfspread` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`cfbyzero` | `cfyield` | `cfprice` | `datetime`

Topics

“Analyzing and Computing Cash Flows” on page 2-11

cfyield

Compute yield to maturity for cash flow given price

Syntax

```
Yield = cfyield(CFlowAmounts,CFlowDates,Price,Settle)
Yield = cfyield( ___ ,Name,Value)
```

Description

`Yield = cfyield(CFlowAmounts,CFlowDates,Price,Settle)` computes yield to maturity for a cash flow given price.

`Yield = cfyield( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

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 = datetime(2003,7,1);
Price = 98;
CFlowAmounts = [30 40 30];
CFlowDates = [datetime(2004,7,15) ; datetime(2005,7,15) ; datetime(2006,7,15)]';
```

Compute the `Yield`.

```
 Yield = cfyield(CFlowAmounts, CFlowDates, Price, Settle)
```

```
Yield = 0.0099
```

Input Arguments

`CFlowAmounts` — Cash flow amounts
vector

Cash flow amounts, specified as an `NINST`-by-`MOSTCFS` matrix. 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.

Data Types: `double`

`CFlowDates` — Cash flow dates
datetime array | string array | date character vector

Cash flow dates, specified as an NINST-by-MOSTCFS matrix using a datetime array, string array, or date character vectors. Each entry contains the date of the corresponding cash flow in CFlowAmounts.

To support existing code, cfyield also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

Price — Prices
vector

Prices specified as an NINST-by-1 vector.

Data Types: double

Settle — Settlement date
datetime array | string array | date character vector

Settlement date, specified as an NMBS-by-1 vector using a datetime array, string array, or date character vectors. The Settle date is the date on which the cash flows are priced.

To support existing code, cfyield also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Note An optional input of size NINST-by-1 is also acceptable as a single value applicable to all contracts. Single values are internally expanded to an array of size NINST-by-1.

Example: Yield = cfyield(CFAmounts,CFDates,Yield,Settle,'Basis',4,'CompoundingFrequency',4)

Basis — Day-count basis
0 (actual/actual) (default) | positive integers of the set [1...13] | vector of positive integers of the set [1...13]

Day-count basis, specified as the comma-separated pair consisting of 'Basis' and a positive integer using a N-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 2-16.

Data Types: `double`

`CompoundingFrequency` — Compounding frequency
2 (actual/actual) (default) | positive integers of the set `[1...13]` | vector of positive integers of the set `[1...13]`

Compounding frequency, specified as the comma-separated pair consisting of `'CompoundingFrequency'` and a positive integer using a `N`-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 2-16.

Data Types: `double`

`CompoundingFrequency` — Compounding frequency
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 the comma-separated pair consisting of `'CompundingFrequency'` and 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

Output Arguments

Yield — Yield for cash flows
vector

Yield for cash flows, returned as an NINST-by-1 vector.

Version History
Introduced in R2012a

R2022b: Serial date numbers not recommended
Not recommended starting in R2022b

Although cfyield supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)

y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also
cfbyzero | cfspread | cfprice | datetime

Topics
“Analyzing and Computing Cash Flows” on page 2-11

cftimes

Time factors corresponding to bond cash flow dates

Syntax

```
TFactors = cftimes(Settle,Maturity)
TFactors = cftimes( ___ ,Name,Value)
```

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( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Compute the Time Factor of a Cash Flow

This example shows how to calculate a cash flow time factor.

```
Settle = datetime(1997,3,15);
Maturity = datetime(1999,9,1);
Period = 2;
TFactors = cftimes(Settle, Maturity, Period)
```

```
TFactors = 1×5

    0.9239    1.9239    2.9239    3.9239    4.9239
```

Input Arguments

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date, specified as an `NINST`-by-`1` vector using a datetime array, string array, or date character vectors. `Settle` must be earlier than `Maturity`.

To support existing code, `cftimes` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date, specified as an `NINST-by-1` vector using a datetime array, string array, or date character vectors. The `Settle` date is the date on which the cash flows are priced.

To support existing code, `cftimes` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `TFactors = cftimes(Settle,Maturity,'Period',4)`

`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 the comma-separated pair consisting of `'Period'` and a 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

`0` (actual/actual) (default) | positive integers of the set `[1...13]` | vector of positive integers of the set `[1...13]`

Day-count basis, specified as the comma-separated pair consisting of `'Basis'` and 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 2-16.

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 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
datetime array | string array | date character vector

Bond Issue date, specified as the comma-separated pair consisting of 'IssueDate' and a scalar or a NUMBONDS-by-1 or 1-by-NUMBONDS vector using a datetime array, string array, or date character vectors.

If you do not specify an IssueDate, the cash flow payment dates are determined from other inputs.

To support existing code, cftimes also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

FirstCouponDate — Irregular or normal first coupon date
datetime array | string array | date character vector

Irregular or normal first coupon date, specified as the comma-separated pair consisting of 'FirstCouponDate' and a scalar or a NUMBONDS-by-1 or 1-by-NUMBONDS vector using a datetime array, string array, or date character vectors.

If you do not specify a FirstCouponDate, the cash flow payment dates are determined from other inputs.

To support existing code, cftimes also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

LastCouponDate — Irregular or normal last coupon date
datetime array | string array | date character vector

Irregular or normal last coupon date, specified as the comma-separated pair consisting of 'LastCouponDate' and a scalar or a NUMBONDS-by-1 or 1-by-NUMBONDS vector using a datetime array, string array, or date character vectors.

If you do not specify a LastCouponDate, the cash flow payment dates are determined from other inputs.

To support existing code, `cftimes` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`StartDate` — Forward starting date of payments
datetime array | string array | date character vector

Forward starting date of payments, specified as the comma-separated pair consisting of `'StartDate'` and a scalar or a `NUMBONDS`-by-`1` or `1`-by-`NUMBONDS` vector using a datetime array, string array, or date character vectors. 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.

To support existing code, `cftimes` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`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 the comma-separated pair consisting of `'CompundingFrequency'` and 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 the comma-separated pair consisting of `'DiscountBasis'` and 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 2-16.

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`

Output Arguments

`TFactors` — Time to cash flow

matrix

Time to cash flow, returned as an `NUMBONDS` rows. 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.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although `cftimes` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Krgin, Dragomir. *Handbook of Global Fixed Income Calculations*. John Wiley & Sons, 2002.

[2] Mayle, Jan. *“Standard Securities Calculations Methods: Fixed Income Securities Formulas for Analytic Measures.”* SIA, Vol 2, Jan 1994.

[3] Stigum, Marcia, and Franklin Robinson. *Money Market and Bond Calculations*. McGraw-Hill, 1996.

See Also

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

Topics

“Analyzing and Computing Cash Flows” on page 2-11

chaikosc

Chaikin oscillator

Syntax

```
chosc = chaikocs(Data)
```

Description

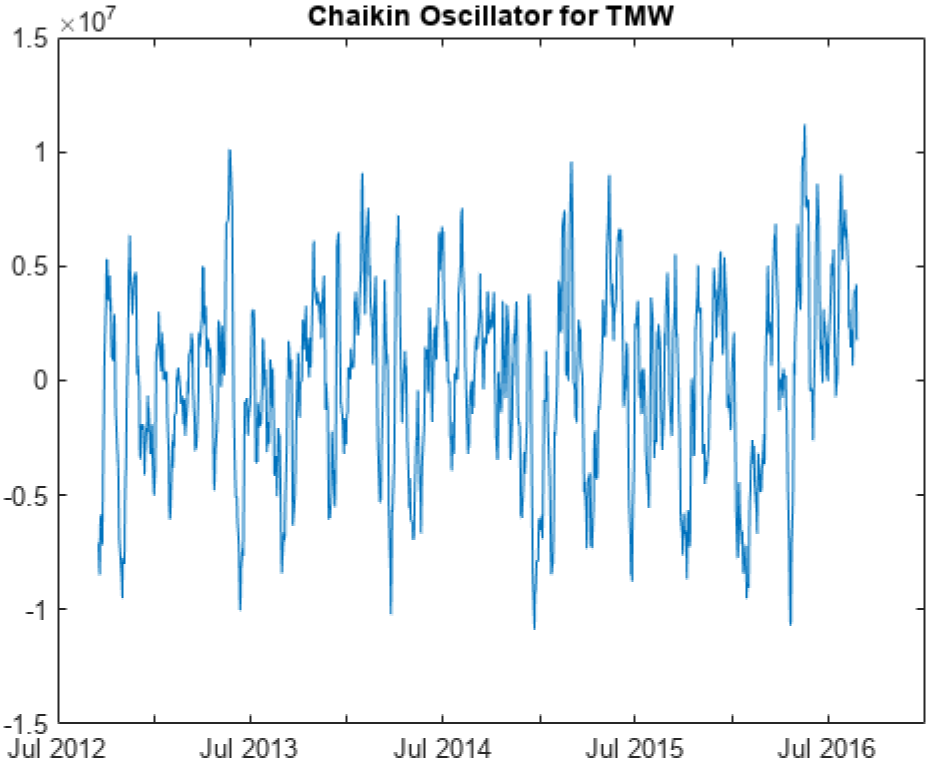
`chosc = chaikocs(Data)` calculates the Chaikin oscillator.

Examples

Calculate the Chaikin Oscillator for a Stock

Load the file `SimulatedStock.mat`, which provides a timetable (`TMW`) for financial data for TMW stock.

```
load SimulatedStock.mat
oscillator = chaikosc(TMW);
plot(oscillator.Time, oscillator.ChaikinOscillator)
title('Chaikin Oscillator for TMW')
```



Input Arguments

Data — Data with high, low, open, close information

matrix | table | timetable

Data with high, low, open, close information, specified as a matrix, table, or timetable. For matrix input, Data is an M-by-4 matrix of high, low, opening, and closing prices. Timetables and tables with M rows must contain variables named 'High', 'Low', 'Open', and 'Close' (case insensitive).

Data Types: double | table | timetable

Output Arguments

chosc — Chaikin oscillator

matrix | table | timetable

Chaikin oscillator, returned with the same number of rows (M) and type (matrix, table, or timetable) as the input Data.

Version History

Introduced before R2006a

R2023a: fints support removed for Data input argument

Behavior changed in R2023a

fints object support for the Data input argument is removed.

R2022b: Support for negative price data

Behavior changed in R2022b

The Data input accepts negative prices.

References

[1] Achelis, S. B. *Technical Analysis from A to Z.* Second Edition. McGraw-Hill, 1995, pp. 91–94.

See Also

adline | timetable | table | chaikvolat

Topics

"Use Timetables in Finance" on page 11-7
"Convert Financial Time Series Objects (fints) to Timetables" on page 11-2

chaikvolat

Chaikin volatility

Syntax

```
volatility = chaikvolat(Data)
volatility = chaikvolat(___,Name,Value)
```

Description

`volatility = chaikvolat(Data)` calculates the Chaikin volatility from a data series of high and low stock prices.

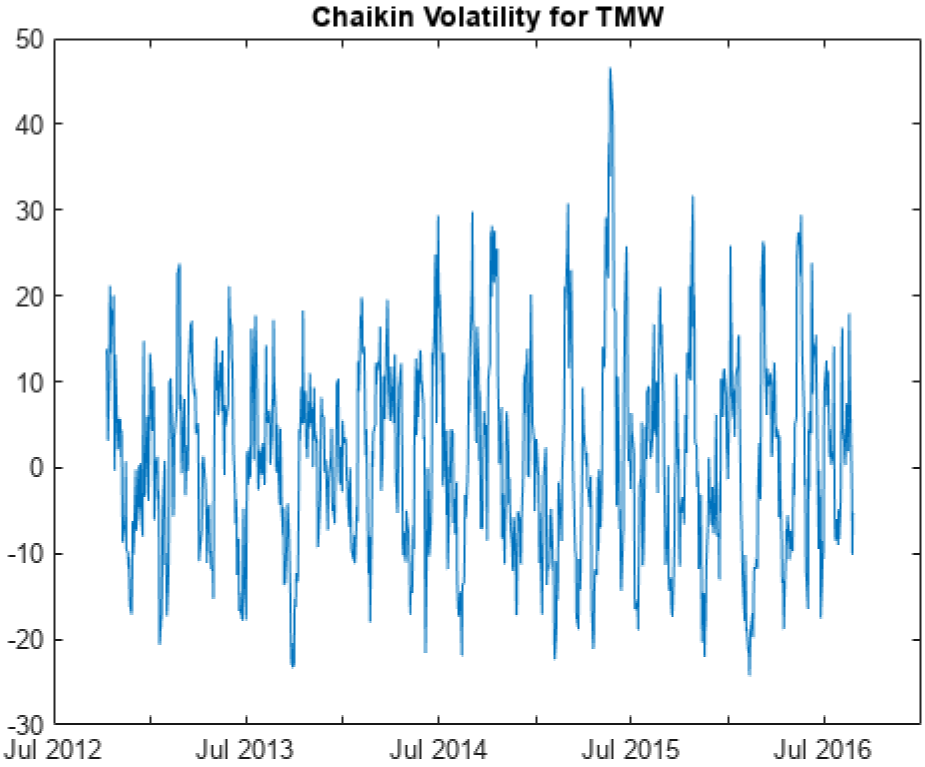
`volatility = chaikvolat(___,Name,Value)` adds optional name-value pair arguments.

Examples

Calculate the Chaikin Volatility for a Stock

Load the file `SimulatedStock.mat`, which provides a timetable (`TMW`) for financial data for TMW stock.

```
load SimulatedStock.mat
volatility = chaikvolat(TMW,'NumPeriods',14,'WindowSize',14);
plot(volatility.Time,volatility.ChaikinVolatility)
title('Chaikin Volatility for TMW')
```



Input Arguments

`Data` — Data with high, low, open, close information

matrix | table | timetable

Data with high, low, open, close information, specified as a vector, matrix, table, or timetable. For matrix input, `Data` is an `M`-by-2 matrix of high and low prices stored in the first and second columns. Timetables and tables with `M` rows must contain variables named `'High'` and `'Low'` (case insensitive).

Data Types: `double` | `table` | `timetable`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `volatility = chaikvolat(TMW,'NumPeriods',10,'WindowSize',10)`

`NumPeriods` — Period difference

`10` (default) | positive integer

Period difference, specified as the comma-separated pair consisting of `'NumPeriods'` and a scalar positive integer.

Data Types: `double`

`WindowSize` — Length of the exponential moving average in periods
`10` (default) | positive integer

Length of the exponential moving average in periods, specified as the comma-separated pair consisting of `'WindowSize'` and a scalar positive integer.

Data Types: `double`

Output Arguments

`volatility` — Chaikin volatility
matrix | table | timetable

Chaikin volatility, returned with the same number of rows (`M`) and the same type (matrix, table, or timetable) as the input `Data`.

More About

Chaikin volatility

Chaikin volatility calculates the Chaikin's volatility from the series of high and low stock prices.

By default, Chaikin's volatility values are based on a 10-period exponential moving average and 10-period difference.

Version History

Introduced before R2006a

R2023a: `fints` support removed for `Data` input argument
Behavior changed in R2023a

`fints` object support for the `Data` input argument is removed.

R2022b: Support for negative price data
Behavior changed in R2022b

The `Data` input accepts negative prices.

References

[1] Achelis, S. B. *Technical Analysis from A to Z.* Second Edition. McGraw-Hill, 1995, pp. 304–305.

See Also

`timetable` | `table` | `chaikosc`

Topics
“Use Timetables in Finance” on page 11-7

“Convert Financial Time Series Objects (fints) to Timetables” on page 11-2

checkFeasibility

Check feasibility of input portfolios against portfolio object

Syntax

```
status = checkFeasibility(obj,pwgt)
```

Description

`status = checkFeasibility(obj,pwgt)` checks the feasibility of input portfolios against a portfolio object.

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 4-17, “PortfolioCVaR Object Workflow” on page 5-16, and “PortfolioMAD Object Workflow” on page 6-15.

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
```

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`

Data Types: `object`

`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`.

Version History

Introduced in R2011a

See Also

`estimateBounds`

Topics

“Validate the Portfolio Problem for Portfolio Object” on page 4-90
“Validate the CVaR Portfolio Problem” on page 5-78
“Validate the MAD Portfolio Problem” on page 6-76
“Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
“Portfolio Optimization Theory” on page 4-3

convert2sur

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

Syntax

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

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.

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
```

Input Arguments

Design — Data series

matrix | cell array

Data series, specified as 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.

Data Types: double | cell

Group — Grouping for data series

vector | cell array

Grouping for data series, specified using separate parameters for each group. Specify groups either by series or by groups:

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

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

Data Types: double | cell

Output Arguments

`DesignSUR` — Seemingly unrelated regression model with a specified grouping of the data series
matrix | cell array

Seemingly unrelated regression model with a specified grouping of the data series, returned as 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.

Version History

Introduced in R2006a

See Also

`ecmnfish` | `mvnrfish`

Topics

“Seemingly Unrelated Regression Without Missing Data” on page 9-17
“Multivariate Normal Linear Regression” on page 9-2

corr2cov

Convert standard deviation and correlation to covariance

Syntax

```
ExpCovariance = corr2cov(ExpSigma)
ExpCovariance = corr2cov( ___ ,ExpCorrC)
```

Description

`ExpCovariance = corr2cov(ExpSigma)` converts standard deviation and correlation to covariance.

`ExpCovariance = corr2cov( ___ ,ExpCorrC)` specifies options using one or more optional arguments in addition to the input arguments in the previous syntax.

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 = 2×2

    0.2500   -0.5000
   -0.5000    4.0000
```

Input Arguments

`ExpSigma` — Standard deviations of each process
vector

Standard deviations of each process, specified as a vector of length `n` with the standard deviations of each process. `n` is the number of random processes.

Data Types: `double`

`ExpCorrC` — Correlation matrix
matrix

(Optional) Correlation matrix, specified as an n-by-n correlation coefficient matrix. A correlation coefficient is a statistic in which the covariance is scaled to a value between minus one (perfect negative correlation) and plus one (perfect positive correlation).

If ExpCorrC is not specified, the processes are assumed to be uncorrelated, and the identity matrix is used.

Data Types: double

Output Arguments

ExpCovariance — Covariance matrix
matrix

Covariance matrix, returned as an n-by-n covariance matrix, where n is the number of processes.

The (*i*,*j*) entry is the expectation of the *i*'th fluctuation from the mean times the *j*'th fluctuation from the mean.

```
ExpCov(i,j) = ExpCorrC(i,j)*ExpSigma(i)*ExpSigma(j)
```

Version History
Introduced before R2006a

See Also
cov2corr | ewstats | nearcorr

cov2corr

Convert covariance to standard deviation and correlation coefficient

Syntax

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

Description

`[ExpSigma,ExpCorrC] = cov2corr(ExpCovariance)` converts covariance to standard deviations and correlation coefficients.

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 = 1×2

    0.5000    2.0000


ExpCorrC = 2×2

    1.0000   -0.5000
   -0.5000    1.0000
```

Input Arguments

ExpCovariance — Covariance matrix
matrix

Covariance matrix, specified as an `n`-by-`n` covariance matrix, where `n` is the number of random processes. For an example, see `cov` or `ewstats`.

Data Types: `double`

Output Arguments

ExpSigma — Standard deviation of each process
vector

Standard deviation of each process, returned as an 1-by-n vector.

The entries of ExpCorrC range from 1 (completely correlated) to -1 (completely anti-correlated). A value of 0 in the (*i*,*j*) entry indicates that the *i*'th and *j*'th processes are uncorrelated.

```
ExpSigma(i) = sqrt( ExpCovariance(i,i) );
ExpCorrC(i,j) = ExpCovariance(i,j)/( ExpSigma(i)*ExpSigma(j) );
```

Data Types: double

ExpCorrC — Correlation coefficients
matrix

Correlation coefficients, returned as an n-by-n matrix.

Version History

Introduced before R2006a

See Also

corr2cov | corrcoef | ewstats | nearcorr

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(datetime(1997,3,14), datetime(2000,11,30), 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 = [datetime(2000,9,30) ; datetime(2001,10,31) ; datetime(2002,11,30)];
NumCouponsRemaining = cpncount(datetime(1997,9,14), Maturity)
```

```
NumCouponsRemaining = 3×1

     7
     9
```

```
    11
```

Input Arguments

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date, specified as a `NBONDS`-by-`1` vector using a datetime array, string array, or date character vectors. `Settle` must be earlier than `Maturity`.

To support existing code, `cpncount` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date, specified as a `NBONDS`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `cpncount` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `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: `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 2-16.

Data Types: `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
datetime array | string array | date character vector

Bond issue date, specified as a `NBONDS`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `cpncount` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`FirstCouponDate` — Date when bond makes first coupon payment
datetime array | string array | date character vector

Date when a bond makes its first coupon payment, specified as a `NBONDS`-by-`1` vector using a datetime array, string array, or date character vectors.

`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.

To support existing code, `cpncount` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`LastCouponDate` — Last coupon date of bond before maturity date
datetime array | string array | date character vector

Last coupon date of a bond before maturity date, specified as a NBONDS-by-1 vector using a datetime array, string array, or date character vectors.

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.

To support existing code, cpncount also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | 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.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although cpncount supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)

y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

accrfrac | cfamounts | cfdates | cftimes | cpndaten | cpndatenq | cpndatep | cpndaysn | cpndaysp | cpnpersz | datetime | cpndatepq

Topics

“Pricing and Computing Yields for Fixed-Income Securities” on page 2-15

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 strings or date character vectors, then `NextCouponDate` is returned as a serial date number. Use the function `datestr` to convert serial date numbers to formatted date character vectors.

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 a datetime array for input arguments.

```
NextCouponDate = cpndaten(datetime(1997,3,14),datetime(2000,11,30), 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'
```

Input Arguments

Settle — Settlement date

datetime array | string array | date character vector

Settlement date, specified as a NUMBONDS-by-1 vector using a datetime array, string array, or date character vectors. Settle must be earlier than Maturity.

To support existing code, cpndaten also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

Maturity — Maturity date

datetime array | string array | date character vector

Maturity date, specified as a NUMBONDS-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, cpndaten also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | 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 2-16.

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

datetime array | string array | date character vector

Bond issue date, specified as a NUMBONDS-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, cpndaten also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

FirstCouponDate — Date when bond makes first coupon payment

datetime array | string array | date character vector

Date when a bond makes its first coupon payment, specified as a NUMBONDS-by-1 vector using a datetime array, string array, or date character vectors.

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.

To support existing code, cpndaten also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

LastCouponDate — Last coupon date of bond before maturity date
datetime array | string array | date character vector

Last coupon date of a bond before maturity date, specified as a NUMBONDS-by-1 vector using a datetime array, string array, or date character vectors.

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.

To support existing code, cpndaten also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | 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 strings or date character vectors, then NextCouponDate is returned as a serial date number. Use the function datestr to convert serial date numbers to formatted date character vectors.

If any of the inputs for Settle, Maturity, IssueDate, FirstCouponDate, and LastCouponDate are datetime arrays, then NextCouponDate is returned as a datetime array.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended
Not recommended starting in R2022b

Although cpndaten supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

accrfrac | cfamounts | cfdates | cftimes | cpncount | cpndatenq | cpndatep | cpndaysn | cpndaysp | cpnpersz | datetime | cpndatepq

Topics

"Pricing and Computing Yields for Fixed-Income Securities" on page 2-15

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 strings or date character vectors, then NextQuasiCouponDate is returned as a serial date number. Use the function datestr to convert serial date numbers to formatted date character vectors.

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 = [datetime(1997,5,30), datetime(1997,12,10)];
Maturity = [datetime(2002,11,30), datetime(2004,6,10)];
```

Compute NextCouponDate for this pair of bonds.

```
NextCouponDate = cpndaten(Settle, Maturity)
NextCouponDate = 2x1 datetime
   31-May-1997
   10-Jun-1998
```

Compute the next quasi coupon dates for these two bonds.

```
NextQuasiCouponDate = cpndatenq(Settle, Maturity)
```

```
NextQuasiCouponDate = 2x1 datetime
   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 = [datetime(1997,11,30), datetime(1998,12,10)];
```

Compute the next coupon dates.

```
NextCouponDate = cpndaten(Settle, Maturity, 2, 0, 1, [],FirstCouponDate)
```

```
NextCouponDate = 2x1 datetime
   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)
```

```
NextQuasiCouponDate = 2x1 datetime
   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.

Input Arguments

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date, specified as a `NUMBONDS`-by-`1` vector using a datetime array, string array, or date character vectors. `Settle` must be earlier than `Maturity`.

To support existing code, `cpndatenq` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date, specified as a `NUMBONDS`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `cpndatenq` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `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 2-16.

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

datetime array | string array | date character vector

Bond issue date, specified as a NUMBONDS-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, cpndatenq also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

FirstCouponDate — Date when bond makes first coupon payment

datetime array | string array | date character vector

Date when a bond makes its first coupon payment, specified as a NUMBONDS-by-1 vector using a datetime array, string array, or date character vectors.

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.

To support existing code, cpndatenq also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

LastCouponDate — Last coupon date of bond before maturity date

datetime array | string array | date character vector

Last coupon date of a bond before maturity date, specified as a NUMBONDS-by-1 vector using a datetime array, string array, or date character vectors.

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.

To support existing code, cpndatenq also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | 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 strings or date character vectors, then NextQuasiCouponDate is returned as a serial date number. Use the function datestr to convert serial date numbers to formatted date character vectors.

If any of the inputs for Settle, Maturity, IssueDate, FirstCouponDate, and LastCouponDate are datetime arrays, then NextQuasiCouponDate is returned as a datetime array.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although cpndatenq supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

accrfrac | cfamounts | cfdates | cftimes | cpncount | cpndaten | cpndatep | cpndatepq | cpndaysn | cpndaysp | cpnpersz | datetime

Topics

“Pricing and Computing Yields for Fixed-Income Securities” on page 2-15

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 strings or date character vectors, then PreviousQuasiCouponDate is returned as a serial date number. Use the function datestr to convert serial date numbers to formatted date character vectors.

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 = [datetime(1997,5,30) ; datetime(1997,12,10)];
Maturity = [datetime(2002,11,30) ; datetime(2004,6,10)];
```

With no FirstCouponDate explicitly supplied, compute the PreviousCouponDate for this pair of bonds.

```
PreviousCouponDate = cpndatep(Settle, Maturity)
PreviousCouponDate = 2x1 datetime
   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 = [datetime(1997,11,30) ; datetime(1998,12,10)];
IssueDate = [datetime(1996,5,30) ; datetime(1996,12,10)];
```

Recompute the `PreviousCouponDate` for this pair of bonds.

```
PreviousCouponDate = cpndatep(Settle, Maturity, 2, 0, 1, IssueDate, FirstCouponDate)
```

```
PreviousCouponDate = 2x1 datetime
   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)
```

```
PreviousQuasiCouponDate = 2x1 datetime
   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`.

Input Arguments

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date, specified as a `NUMBONDS`-by-`1` vector using a datetime array, string array, or date character vectors. `Settle` must be earlier than `Maturity`.

To support existing code, `cpndatepq` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date, specified as a NUMBONDS-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, cpndatepq also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | 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 2-16.

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
datetime array | string array | date character vector

Bond issue date, specified as a `NUMBONDS`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `cpndatepq` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`FirstCouponDate` — Date when bond makes first coupon payment
datetime array | string array | date character vector

Date when a bond makes its first coupon payment, specified as a `NUMBONDS`-by-`1` vector using a datetime array, string array, or date character vectors.

`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.

To support existing code, `cpndatepq` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`LastCouponDate` — Last coupon date of bond before maturity date
datetime array | string array | date character vector

Last coupon date of a bond before maturity date, specified as a `NUMBONDS`-by-`1` vector using a datetime array, string array, or date character vectors.

`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.

To support existing code, `cpndatepq` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `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 strings or date character vectors, then PreviousQuasiCouponDate is returned as a serial date number. Use the function datestr to convert serial date numbers to formatted date character vectors.

If any of the inputs for Settle, Maturity, IssueDate, FirstCouponDate, and LastCouponDate are datetime arrays, then PreviousQuasiCouponDate is returned as a datetime array.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although cpndatepq supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

accrfrac | cfamounts | cfdates | cftimes | cpncount | cpndaten | cpndatep | cpndatenq | cpndaysn | cpndaysp | cpnpersz | datetime

Topics

"Pricing and Computing Yields for Fixed-Income Securities" on page 2-15

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 strings or date character vectors, then `PreviousCouponDate` is returned as a serial date number. Use the function `datestr` to convert serial date numbers to formatted date character vectors.

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(datetime(1997,3,14),datetime(2000,6,30),2, 0, 0)
```

```
PreviousCouponDate = datetime
   30-Dec-1996
```

Determine the `PreviousCouponDate` when using datetime arrays for input arguments.

```
PreviousCouponDate = cpndatep(datetime(1997,3,14),datetime(2000,6,30),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(datetime(1997,3,14),datetime(2000,6,30),2, 0, 1)
```

```
PreviousCouponDate = datetime
   31-Dec-1996
```

Determine the `PreviousCouponDate` when using an input vector for `Maturity`.

```
Maturity = [datetime(2000,4,30) ; datetime(2000,5,31) ; datetime(2000,6,30)];
PreviousCouponDate = cpndatep(datetime(1997,3,14), Maturity)
```

```
PreviousCouponDate = 3x1 datetime
   31-Oct-1996
   30-Nov-1996
   31-Dec-1996
```

Input Arguments

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date, specified as a `NUMBONDS`-by-`1` vector using a datetime array, string array, or date character vectors. `Settle` must be earlier than `Maturity`.

To support existing code, `cpndatep` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date, specified as a `NUMBONDS`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `cpndatep` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `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 2-16.

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

datetime array | string array | date character vector

Bond issue date, specified as a NUMBONDS-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, cpndatep also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

FirstCouponDate — Date when bond makes first coupon payment

datetime array | string array | date character vector

Date when a bond makes its first coupon payment, specified as a NUMBONDS-by-1 vector using a datetime array, string array, or date character vectors.

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.

To support existing code, cpndatep also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

LastCouponDate — Last coupon date of bond before maturity date

datetime array | string array | date character vector

Last coupon date of a bond before maturity date, specified as a NUMBONDS-by-1 vector using a datetime array, string array, or date character vectors.

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.

To support existing code, cpndatep also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | 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 strings or date character vectors, then PreviousCouponDate is returned as a serial date number. Use the function datestr to convert serial date numbers to formatted date character vectors.

If any of the inputs for Settle, Maturity, IssueDate, FirstCouponDate, and LastCouponDate are datetime arrays, then PreviousCouponDate is returned as a datetime array.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although `cpndatep` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

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 2-15

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 `NUMBONDS`-by-`1` vector containing the number of days to the next coupon date.

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.

Examples

Calculate the Number of Days From Settlement Date to Next Coupon Date

Determine the `NumDaysNext` when using datetimes for input arguments.

```
NumDaysNext = cpndaysn(datetime(2000,9,14) , datetime(2001,6,30), 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 a datetime array for `Maturity`.

```
Maturity = [datetime(2001,4,30) ; datetime(2001,5,31) ; datetime(2001,6,30)];
NumDaysNext = cpndaysn(datetime(2000,9,14), Maturity)
```

```
NumDaysNext = 3×1

    47
    77
   108
```

Input Arguments

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date, specified as a `NUMBONDS`-by-`1` vector using a datetime array, string array, or date character vectors. `Settle` must be earlier than `Maturity`.

To support existing code, `cpndaysn` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date, specified as a `NUMBONDS`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `cpndaysn` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `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 2-16.

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
datetime array | string array | date character vector

Bond issue date, specified as a `NUMBONDS`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `cpndaysn` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`FirstCouponDate` — Date when bond makes first coupon payment
datetime array | string array | date character vector

Date when a bond makes its first coupon payment, specified as a `NUMBONDS`-by-`1` vector using a datetime array, string array, or date character vectors.

`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.

To support existing code, `cpndaysn` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`LastCouponDate` — Last coupon date of bond before maturity date
datetime array | string array | date character vector

Last coupon date of a bond before maturity date, specified as a NUMBONDS-by-1 vector using a datetime array, string array, or date character vectors.

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.

To support existing code, cpndaysn also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | 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.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended
Not recommended starting in R2022b

Although cpndaysn supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

accrfrac | cfamounts | cfdates | cftimes | cpncount | cpndaten | cpndatep | cpndatenq | cpndaysp | cpnpersz | datetime | cpndatepq

Topics

“Pricing and Computing Yields for Fixed-Income Securities” on page 2-15

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 `NUMBONDS`-by-`1` vector containing the number of days from the previous coupon date to settlement.

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.

Examples

Calculate the Number of Days Between Previous Coupon Date and Settlement Date

Determine the `NumDaysPrevious` when using datetimes for input arguments.

```
NumDaysPrevious = cpndaysp(datetime(2000,3,14),datetime(2001,6,30), 2, 0, 0)
```

```
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 a datetime array for `Maturity`.

```
Maturity = [datetime(2001,4,30) ; datetime(2001,5,31) ; datetime(2001,6,30)];
NumDaysPrevious = cpndaysp(datetime(2000,3,14), Maturity)
```

```
NumDaysPrevious = 3×1
```

```
   135
   105
    74
```

Input Arguments

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date, specified as a `NUMBONDS`-by-`1` vector using a datetime array, string array, or date character vectors. `Settle` must be earlier than `Maturity`.

To support existing code, `cpndaysp` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date, specified as a `NUMBONDS`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `cpndaysp` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `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 2-16.

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
datetime array | string array | date character vector

Bond issue date, specified as a `NUMBONDS`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `cpndaysp` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`FirstCouponDate` — Date when bond makes first coupon payment
datetime array | string array | date character vector

Date when a bond makes its first coupon payment, specified as a `NUMBONDS`-by-`1` vector using a datetime array, string array, or date character vectors.

`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.

To support existing code, `cpndaysp` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`LastCouponDate` — Last coupon date of bond before maturity date
datetime array | string array | date character vector

Last coupon date of a bond before maturity date, specified as a NUMBONDS-by-1 vector using a datetime array, string array, or date character vectors.

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.

To support existing code, cpndaysp also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | 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.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although cpndaysp supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

accrfrac | cfamounts | cfdates | cftimes | cpncount | cpndaten | cpndatenq | cpndatep | cpndaysn | cpndatepq | cpnpersz | datetime

Topics

“Pricing and Computing Yields for Fixed-Income Securities” on page 2-15

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 `NUMBONDS`-by-`1` vector containing the number of days in the coupon period containing the settlement date.

.

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.

Examples

Calculate the Number of Days in the Coupon Period Containing Settlement Date

Determine the `NumDaysPeriod` when using datetimes for input arguments.

```
NumDaysPeriod = cpnpersz(datetime(2000,9,14),datetime(2001,6,30), 2, 0, 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 = [datetime(2001,6,30) ; datetime(2001,5,31) ; datetime(2001,6,30)];
NumDaysPeriod = cpnpersz('14-Sep-2000', Maturity)
```

```
NumDaysPeriod = 3×1
```

```
   184
   183
   184
```

Input Arguments

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date, specified as a `NUMBONDS`-by-`1` vector using a datetime array, string array, or date character vectors. `Settle` must be earlier than `Maturity`.

To support existing code, `cpnpersz` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date, specified as a `NUMBONDS`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `cpnpersz` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `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 2-16.

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
datetime array | string array | date character vector

Bond issue date, specified as a `NUMBONDS`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `cpnpersz` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`FirstCouponDate` — Date when bond makes first coupon payment
datetime array | string array | date character vector

Date when a bond makes its first coupon payment, specified as a `NUMBONDS`-by-`1` vector using a datetime array, string array, or date character vectors.

`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.

To support existing code, `cpnpersz` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`LastCouponDate` — Last coupon date of bond before maturity date
datetime array | string array | date character vector

Last coupon date of a bond before maturity date, specified as a NUMBONDS-by-1 vector using a datetime array, string array, or date character vectors.

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.

To support existing code, cpnpersz also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | 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.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although cpnpersz supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)

y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

accrfrac | cfamounts | cfdates | cftimes | cpncount | cpndaten | cpndatenq | cpndatep | cpndaysn | cpndatepq | cpndaysp | datetime

Topics
“Pricing and Computing Yields for Fixed-Income Securities” on page 2-15

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 https://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)
```

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`

Version History

Introduced in R2007b

See Also

`holidays`

Topics

“Handle and Convert Dates” on page 2-2
“Trading Calendars User Interface” on page 12-2
“UICalendar User Interface” on page 12-4

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.

Version History

Introduced before R2006a

See Also

`cur2str` | `frac2cur`

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.

Version History

Introduced before R2006a

See Also

frac2cur | cur2frac

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(datetime(2015,7,31),datetime(2015,9,30), 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.

Input Arguments

`Settle` — Settlement date

datetime scalar | string scalar | date character vector

Settlement date, specified as a scalar datetime, string, or date character vector.

To support existing code, `date2time` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date, specified as a scalar or `N`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `date2time` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `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 2-16.

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.

More About

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
```

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although `date2time` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`cftimes` | `disc2rate` | `rate2disc` | `time2date` | `yearfrac` | `datetime`

Topics

"Handle and Convert Dates" on page 2-2

dateaxis

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

Syntax

```
dateaxis(Tickaxis,DateForm,StartDate)
```

Description

`dateaxis(Tickaxis,DateForm,StartDate)` replaces axis tick labels with date labels.

Examples

Replace Axis Tick Labels with Date Labels

This example shows how to use dateaxis to replaces axis tick labels with date labels on a graphic figure.

Convert the *x*-axis labels to an automatically determined date format.

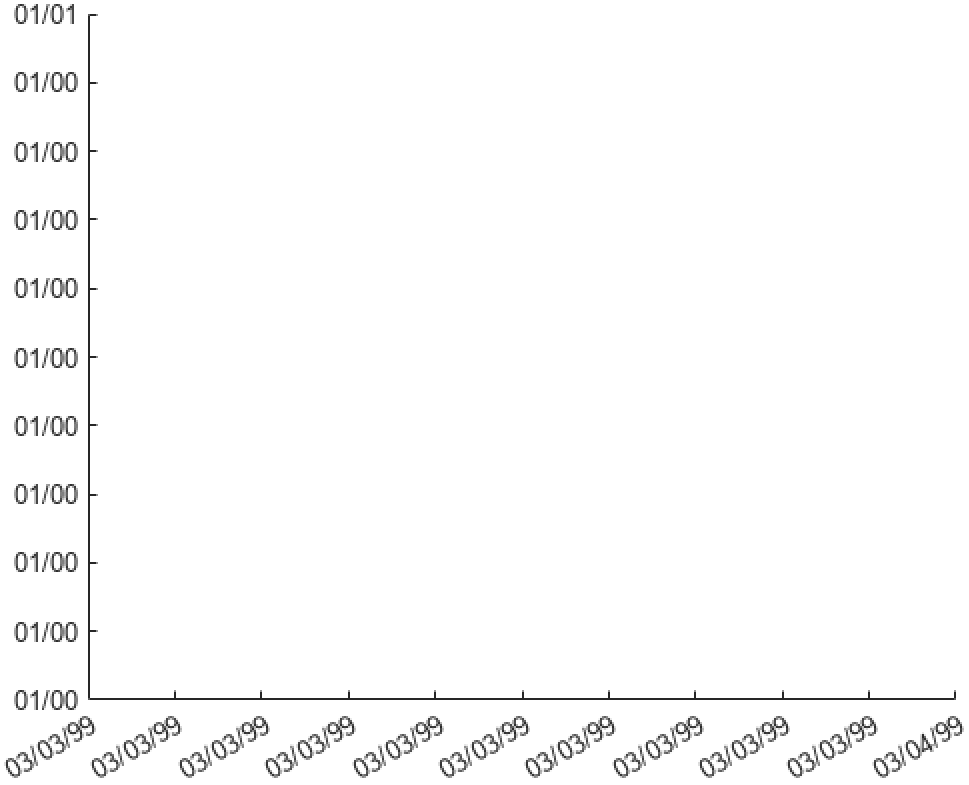
```
dateaxis('y', 6)
```

Convert the *y*-axis labels to the month/day format.

```
dateaxis('x', 2, datetime(1999,3,3))
```

Convert the *x*-axis labels to the month/day/year format. The minimum *x*-tick value is treated as March 3, 1999.

```
dateaxis('x', 2, datetime(1999,3,3))
```



Input Arguments

`Tickaxis` — Determines which axis tick labels, x, y, z to replace

`'x'` (default) | character vector with value `'x'`, `'y'`, or `'z'`

(Optional) Determines which axis tick labels —*x*, *y*, or *z*— to replace, specified as a character vector.

Data Types: `char`

`DateForm` — Defines which date format to use

date format based on the span of the axis limits (default) | integer from `0` to `17`

(Optional) Defines which date format to use, specified 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`).

DateForm	Format	Description
0	`01-Mar-1999 15:45:17`	day-month-year hour:minute:second
1	`01-mar-1999`	day-month-year
2	`03/01/99`	month/day/year
3	`Mar`	month, three letters

DateForm	Format	Description
4	M	month, single letter
5	3	month
6	03/01	month/day
7	1	day of month
8	Wed	day of week, three letters
9	W	day of week, single letter
10	1999	year, four digits
11	99	year, two digits
12	Mar99	month year
13	15:45:17	hour:minute:second
14	03:45:17 PM	hour:minute:second AM or PM
15	15:45	hour:minute
16	03:45 PM	hour:minute AM or PM
17	95/03/01	year month day

Refer to the MATLAB `set` command for information on modifying the axis tick values and other axis parameters.

Data Types: `double`

`StartDate` — Assigns the date to the first axis tick value
lower axis limit converted to the appropriate date number (default) | datetime scalar | string scalar | date character vector

(Optional) Assigns the date to the first axis tick value, specified as a scalar datetime, string, or date character vector.

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.

To support existing code, `dateaxis` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although `dateaxis` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)

y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

bollinger | candle | datenum | datestr | highlow | movavg | pointfig | datetime

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
```

Input Arguments

NumMat — Numeric matrix to display
serial date numbers

Numeric matrix to display, specified as 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.

Version History

Introduced before R2006a

See Also

`datenum` | `datestr`

Topics

"Handle and Convert Dates" on page 2-2

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 = datetime(1999,7,1:31);
Subset = [datetime(1999,7,10) ; datetime(1999,7,20)];
Indices = datefind(Subset, Superset, 1)
```

```
Indices = 6×1

     9
    10
    11
    19
    20
    21
```

Input Arguments

`Subset` — Subset of dates to find matching dates
datetime array | string array | date character vector

Subset of dates to find matching dates in `Superset`, specified as a matrix using a datetime array, string array, or date character vectors.

`Subset` and `Superset` can be either be a datetime array, string array, or date character vectors. 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 = [datetime(1997,7,10); datetime(1997,7,20)];`

To support existing code, `datefind` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Superset` — Superset of dates

datetime array | string array | date character vector

Superset of dates, specified as a matrix of using a datetime array, string array, or date character vectors, whose elements are sought.

`Subset` and `Superset` can be either a datetime array, string array, or date character vectors. 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 = datetime(1997,7,1:31);`

To support existing code, `datefind` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`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.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although datefind supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

datenum | datetime

Topics

“Handle and Convert Dates” on page 2-2

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 datetime, then `NumberMonths` must be an `N`-by-`1` vector of integers or a single integer. `TargetDate` is then an `N`-by-`1` vector of datetimes.

If `StartDate` is a string or date character vector, `TargetDate` is returned as a serial date number. Use the function `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 datetime for `StartDate`.

```
StartDate = datetime(1997,6,3);
NumberMonths = 6;
DayFlag = 0;
Basis = 0;
EndMonthRule = 1;

TargetDate = datemnth(StartDate, NumberMonths, DayFlag,Basis, EndMonthRule)
```

```
TargetDate = datetime
   03-Dec-1997
```

Determine the `TargetDate` in a future month using a vector for `NumberMonths`.

```
NumberMonths = [1; 3; 5; 7; 9];
TargetDate = datemnth(datetime(2001,1,31), NumberMonths)
TargetDate = 5x1 datetime
   28-Feb-2001
   30-Apr-2001
   30-Jun-2001
   31-Aug-2001
   31-Oct-2001
```

Input Arguments

StartDate — Start date

datetime array | string array | date character vector

Start date, specified as an N-by-1 or 1-by-N vector using a datetime array, string array, or date character vectors.

To support existing code, datemnth also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | 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 values0, 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 2-16.

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.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended
Not recommended starting in R2022b

Although `datemnth` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`datestr` | `datevec` | `days360` | `days365` | `daysact` | `daysdif` | `wrkdydif` | `datetime`

Topics

"Handle and Convert Dates" on page 2-2
"Trading Calendars User Interface" on page 12-2
"UICalendar User Interface" on page 12-4

datewrkdy

Date of future or past workday

Syntax

```
EndDate = datewrkdy(StartDate,NumberWorkDays,NumberHolidays)
```

Description

`EndDate = datewrkdy(StartDate,NumberWorkDays,NumberHolidays)` returns the datetime 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 datetime, then `NumberWorkDays` must be an `N`-by-`1` vector of integers or a single integer. `EndDate` is then an `N`-by-`1` vector of datetimes.

If `StartDate` is a string or date character vector, `EndDate` is returned as a serial date number. Use the function `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 for `StartDate`.

```
EndDate = datewrkdy(datetime(2000,12,12), 16, 2)
```

```
EndDate = datetime
   04-Jan-2001
```

Determine the `EndDate` for future workdays using a vector for `NumberWorkDays`.

```
NumberWorkDays = [16; 20; 44];
EndDate = datewrkdy(datetime(2000,12,12), NumberWorkDays, 2)
```

```
EndDate = 3x1 datetime
   04-Jan-2001
   10-Jan-2001
   13-Feb-2001
```

Input Arguments

StartDate — Start date

datetime array | string array | date character vector

Start date, specified as an N-by-1 or 1-by-N vector using a datetime array, string array, or date character vectors.

To support existing code, datewrkdy also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | 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 for the future or past date.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended
Not recommended starting in R2022b

Although `datewrkdy` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`busdate` | `holidays` | `isbusday` | `wrkdydif` | `datetime`

Topics

“Handle and Convert Dates” on page 2-2
“Trading Calendars User Interface” on page 12-2
“UICalendar User Interface” on page 12-4

days252bus

Number of business days between dates

Syntax

```
NumberDays = days252bus(StartDate,EndDate)
NumberDays = days252bus( ___ HolidayVector)
```

Description

`NumberDays = days252bus(StartDate,EndDate)` computes the number of business days (that is, non-holiday or non-weekend) between the two input dates.

`NumberDays = days252bus( ___ HolidayVector)` adds an optional argument for `HolidayVector`. If a holiday vector is not optionally specified, then the `holidays.m` file is used to determine the holidays.

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(datetime(2009,1,1), datetime(2009,8,1))
```

```
NumberDays = 146
```

Input Arguments

`StartDate` — Start date
datetime array | string array | date character vector

Start date, specified as a scalar or an `N`-by-`1` or `1`-by-`N` vector using a datetime array, string array, or date character vectors.

To support existing code, `days252bus` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `double` | `char` | `string` | `datetime`

`EndDate` — End date
datetime array | string array | date character vector

End date, specified as a scalar or an `N`-by-`1` or `1`-by-`N` vector using a datetime array, string array, or date character vectors.

To support existing code, `days252bus` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

HolidayVector — Holidays
holidays.m (default) | datetime array | string array | date character vector | serial date number

Holidays, specified as a scalar or an N-by-1 or 1-by-N vector using a datetime array, string array, or date character vectors.

To support existing code, days252bus also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

Output Arguments

NumberDays — 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.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended
Not recommended starting in R2022b

Although days252bus supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

days360psa | daysact | daysdif | days365 | datetime

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

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 datetime, 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 for `StartDate` and `EndDate`.

```
NumDays = days360(datetime(2000,1,15) , datetime(2000,3,15))
```

```
NumDays = 60
```

Determine the `NumDays` using a datetime array for `EndDate`.

```
MoreDays = [datetime(2000,3,15) ; datetime(2000,4,15) ; datetime(2000,6,15)];
NumDays = days360(datetime(2000,1,15), MoreDays)
```

```
NumDays = 3×1

    60
    90
   150
```

Input Arguments

`StartDate` — Start date
datetime array | string array | date character vector

Start date, specified as a scalar or an N-by-1 or 1-by-N vector using a datetime array, string array, or date character vectors.

To support existing code, `days360` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`EndDate` — End date
datetime array | string array | date character vector

End date, specified as a scalar or an N-by-1 or 1-by-N vector using a datetime array, string array, or date character vectors.

To support existing code, `days360` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `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.

Version History
Introduced before R2006a

R2022b: Serial date numbers not recommended
Not recommended starting in R2022b

Although `days360` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)

y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

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

See Also

`days365` | `daysact` | `daysdif` | `wrkdydif` | `yearfrac` | `datetime`

Topics

"Handle and Convert Dates" on page 2-2

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 datetime, 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 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 datetimes for `StartDate` and `EndDate`.

```
NumDays = days360e(datetime(2002,1,1), datetime(2002,2,1))
```

```
NumDays = 30
```

Determine the `NumDays` using a datatime arrayr for `EndDate`.

```
MoreDays = [datetime(2000,3,15) ; datetime(2000,4,15) ; datetime(2000,6,15)];
NumDays = days360e(datetime(2000,1,15), MoreDays)
```

```
NumDays = 3×1

    60
    90
   150
```

Input Arguments

StartDate — Start date

datetime array | string array | date character vector

Start date, specified as a scalar or an N-by-1 or 1-by-N vector using a datetime array, string array, or date character vectors.

To support existing code, days360e also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

EndDate — End date

datetime array | string array | date character vector

End date, specified as a scalar or an N-by-1 or 1-by-N vector using a datetime array, string array, or date character vectors.

To support existing code, days360e also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | 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.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although days360e supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

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

See Also

`days360` | `days360isda` | `days360psa` | `datetime`

Topics

"Handle and Convert Dates" on page 2-2

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 datetime, 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 datetimes for `StartDate` and `EndDate`.

```
NumDays = days360isda(datetime(2002,1,1), datetime(2002,2,1))
```

```
NumDays = 30
```

Determine the `NumDays` using a datatime array for `EndDate`.

```
MoreDays = [datetime(2000,3,15) ; datetime(2000,4,15) ; datetime(2000,6,15)];
NumDays = days360isda(datetime(2000,1,15), MoreDays)
```

```
NumDays = 3×1

    60
    90
   150
```

Input Arguments

StartDate — Start date

datetime array | string array | date character vector

Start date, specified as a scalar or an N-by-1 or 1-by-N vector using a datetime array, string array, or date character vectors.

To support existing code, days360isda also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

EndDate — End date

datetime array | string array | date character vector

End date, specified as a scalar or an N-by-1 or 1-by-N vector using a datetime array, string array, or date character vectors.

To support existing code, days360isda also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | 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.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although days360isda supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

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

See Also

`days360` | `days360e` | `datetime`

Topics

"Handle and Convert Dates" on page 2-2

days360psa

Days between dates based on 360-day year (Public Securities Association (PSA) compliant)

Syntax

```
NumDays = days360psa(StartDate,EndDate)
```

Description

`NumDays = days360psa(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 datetime, 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 datetimes for `StartDate` and `EndDate`.

```
NumDays = days360psa(datetime(2002,1,1) , datetime(2002,2,1))
```

```
NumDays = 30
```

Determine the `NumDays` using a datetime array for `EndDate`.

```
MoreDays = [datetime(2000,3,15) ; datetime(2000,4,15) ; datetime(2000,6,15)];
NumDays = days360psa(datetime(2000,1,15), MoreDays)
```

```
NumDays = 3×1

    60
    90
   150
```

Input Arguments

StartDate — Start date

datetime array | string array | date character vector

Start date, specified as a scalar or an N-by-1 or 1-by-N vector using a datetime array, string array, or date character vectors.

To support existing code, days360psa also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

EndDate — End date

datetime array | string array | date character vector

End date, specified as a scalar or an N-by-1 or 1-by-N vector using a datetime array, string array, or date character vectors.

To support existing code, days360psa also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | 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.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although days360psa supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

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

See Also

`days360` | `days360e` | `days360isda` | `datetime`

Topics

"Handle and Convert Dates" on page 2-2

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 datetime, 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 datetimes for `StartDate` and `EndDate`.

```
NumDays = days365(datetime(2000,1,15) , datetime(2000,3,15))
```

```
NumDays = 59
```

Determine the `NumDays` using a datetime array for `EndDate`.

```
MoreDays = [datetime(2000,3,15) ; datetime(2000,4,15) ; datetime(2000,6,15)];
NumDays = days365('15-jan-2000', MoreDays)
```

```
NumDays = 3×1

    59
    90
   151
```

Input Arguments

StartDate — Start date

datetime array | string array | date character vector

Start date, specified as a scalar or an N-by-1 or 1-by-N vector using a datetime array, string array, or date character vectors.

To support existing code, days365 also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

EndDate — End date

datetime array | string array | date character vector

End date, specified as a scalar or an N-by-1 or 1-by-N vector using a datetime array, string array, or date character vectors.

To support existing code, days365 also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | 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.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although days365 supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

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

See Also

`days360` | `daysact` | `daysdif` | `wrkdydif` | `yearfrac` | `datetime`

Topics

"Handle and Convert Dates" on page 2-2

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 datetime, 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 datetimes for `StartDate` and `EndDate`.

```
NumDays = daysact(datetime(2002,9,7) , datetime(2002,12,25))
```

```
NumDays = 109
```

Determine the `NumDays` using a datetime array for `EndDate`.

```
MoreDays = [datetime(2002,9,7) ; datetime(2002,10,22) ; datetime(2002,11,5)];
NumDays = daysact(MoreDays, '12/25/2002')
```

```
NumDays = 3×1

   109
    64
    50
```

Input Arguments

StartDate — Start date

datetime array | string array | date character vector

Start date, specified as a scalar or an N-by-1 or 1-by-N vector using a datetime array, string array, or date character vectors.

To support existing code, daysact also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

EndDate — End date

datetime array | string array | date character vector

End date, specified as a scalar or an N-by-1 or 1-by-N vector using a datetime array, string array, or date character vectors.

To support existing code, daysact also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | 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.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although daysact supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

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` | `datevec` | `days360` | `days365` | `daysdif` | `datetime`

Topics

"Handle and Convert Dates" on page 2-2
"Trading Calendars User Interface" on page 12-2
"UICalendar User Interface" on page 12-4

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 string or date character vector, `NewDate` is returned as a serial 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 string or date character vector, `NewDate` is returned as a serial date number. Use the function `datestr` to convert serial date numbers to formatted date character vectors.

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 for `StartDate`.

```
NewDate = daysadd(datetime(2004,2,1), 31)
```

```
NewDate = datetime
   03-Mar-2004
```

Determine the `NewDate` using a datetime arraay for `StartDate`.

```
MoreDays = [datetime(2002,9,7) ; datetime(2002,10,22) ; datetime(2002,11,5)];
NewDate = daysadd(MoreDays, 31 ,2)
NewDate = 3x1 datetime
   08-Oct-2002
   22-Nov-2002
   06-Dec-2002
```

Input Arguments

StartDate — Start date

datetime array | string array | date character vector

Start date, specified as a scalar or an N-by-1 or 1-by-N vector using a datetime array, string array, or date character vectors.

To support existing code, daysadd also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | 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 2-16.

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: 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 string or date character vector, NewDate is returned as a serial date number. Use the function datestr to convert serial date numbers to formatted date character vectors.

If StartDate is a datetime array, then NewDate is returned as a datetime array.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although daysadd supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

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

See Also

daysdif | datetime

Topics

"Handle and Convert Dates" on page 2-2
"Trading Calendars User Interface" on page 12-2

“UICalendar User Interface” on page 12-4

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 cell array of character vectors , then `EndDate` must be an *n*-row cell array of character vectors or a single date character vector. `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 datetime array, then `EndDate` must be an *n*-row datetime array or a single datetime. `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 datetimes for `StartDate` and `EndDate`.

```
NumDays = daysdif(datetime(1999,3,1) ,datetime(2000,3,1), 1)
```

```
NumDays = 360
```

Determine the `NumDays` using a datetime array for `EndDate`.

```
MoreDays = [datetime(2001,3,1) ; datetime(2002,3,1) ; datetime(2003,3,1)];
NumDays = daysdif(datetime(1998,3,1), MoreDays)
```

```
NumDays = 3×1
```

```
    1096
    1461
    1826
```

Input Arguments

StartDate — Start date

datetime array | string array | date character vector

Start date, specified as a scalar or an N-by-1 or 1-by-N vector using a datetime array, string array, or date character vectors.

To support existing code, daysdif also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

EndDate — End date

datetime array | string array | date character vector

End date, specified as a scalar or an N-by-1 or 1-by-N vector using a datetime array, string array, or date character vectors.

To support existing code, daysdif also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | 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 2-16.

Data Types: `double`

Output Arguments

`NumDays` — Number of days between dates `StartDate` and `EndDate`
integer

Number of days between the `StartDate` and `EndDate`. `NumDays` is returned as an integer.

Note The first date for `StartDate` is not included when determining the number of days between first and last date.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although `daysdif` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)

y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

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

See Also

`dec2thirtytwo` | `datetime`

Topics

“Handle and Convert Dates” on page 2-2

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 = 2×1

   101
   102


Fractions = 2×1

    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`.

Version History

Introduced before R2006a

See Also

`thirtytwo2dec`

depfixdb

Fixed declining-balance depreciation schedule

Syntax

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

Description

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

`Depreciation = depfixdb( ___ ,Month)` adds an optional argument.

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 = 1×5
10³ ×

    2.4251    1.8904    1.4737    1.1488    0.8955
```

Input Arguments

`Cost` — Initial value of the asset
numeric

initial value of the asset, specified as a scalar numeric.

Data Types: `double`

`Salvage` — Salvage value of the asset
numeric

Salvage value of the asset, specified as a scalar numeric.

Data Types: `double`

`Life` — Life of the asset in years
numeric

life of the asset in years, specified as scalar numeric.

Data Types: `double`

`Period` — Number of years to calculate

integer

Number of years to calculate, specified as scalar integer.

Data Types: `double`

`Month` — Number of months in the first year of asset life

`12` (default) | numeric

(Optional) Number of months in the first year of asset life, specified as a scalar numeric.

Data Types: `double`

Output Arguments

`Depreciation` — Depreciation

vector

Depreciation, returned as the fixed declining-balance depreciation for each `Period`.

Version History

Introduced before R2006a

See Also

`depgendb` | `deprdv` | `depsoyd` | `depstln`

Topics

"Analyzing and Computing Cash Flows" on page 2-11

depgendb

General declining-balance depreciation schedule

Syntax

```
Depreciation = depgendb(Cost,Salvage,Life,Factor)
```

Description

`Depreciation = depgendb(Cost,Salvage,Life,Factor)` computes 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 = 1×5
10^3 ×

    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`.

Input Arguments

`Cost` — Initial value of the asset
numeric

initial value of the asset, specified as a scalar numeric.

Data Types: `double`

`Salvage` — Salvage value of the asset

numeric

Salvage value of the asset, specified as a scalar numeric.

Data Types: `double`

`Life` — Number of periods over which the asset is depreciated

numeric

Number of periods over which the asset is depreciated, specified as a scalar numeric.

Data Types: `double`

`Factor` — Depreciation factor

numeric

Depreciation factor, specified as a scalar numeric. When `Factor` = `2`, then the double-declining-balance method is used.

Data Types: `double`

Output Arguments

`Depreciation` — Depreciation

vector

Depreciation, returned as the declining-balance depreciation for each period.

Version History

Introduced before R2006a

See Also

`depfixdb` | `deprdv` | `depsoyd` | `depstln`

Topics

"Analyzing and Computing Cash Flows" on page 2-11

deprdv

Remaining depreciable value

Syntax

```
Depreciation = deprdv(Cost,Salvage,Accum)
```

Description

`Depreciation = deprdv(Cost,Salvage,Accum)` computes 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
```

Input Arguments

`Cost` — Initial value of the asset
numeric

Initial value of the asset, specified as a scalar numeric.

Data Types: `double`

`Salvage` — Salvage value of the asset
numeric

Salvage value of the asset, specified as a scalar numeric.

Data Types: `double`

`Accum` — Accumulated depreciation of the asset for prior periods
numeric

Accumulated depreciation of the asset for prior periods, specified as a scalar numeric.

Data Types: `double`

Output Arguments

`Depreciation` — Depreciation

numeric

Depreciation, returned as the remaining depreciable value for an asset.

Version History

Introduced before R2006a

See Also

`depfixdb` | `depgendb` | `depsoyd` | `depstln`

Topics

"Analyzing and Computing Cash Flows" on page 2-11

depsoyd

Sum of years' digits depreciation

Syntax

```
Sum = depsoyd(Cost,Salvage,Life)
```

Description

`Sum = depsoyd(Cost,Salvage,Life)` computes the depreciation for an asset using the sum of years' digits method.

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 = 10×1
10^3 ×

    2.1818
    1.9636
    1.7455
    1.5273
    1.3091
    1.0909
    0.8727
    0.6545
    0.4364
    0.2182
```

Input Arguments

`Cost` — Initial value of the asset

numeric

initial value of the asset, specified as a scalar numeric.

Data Types: `double`

`Salvage` — Salvage value of the asset

numeric

Salvage value of the asset, specified as a scalar numeric.

Data Types: `double`

`Life` — Depreciable life of the asset in years

numeric

Depreciable life of the asset in years, specified as a scalar numeric.

Data Types: `double`

Output Arguments

`Sum` — Depreciation values

vector

Depreciation values, returned as a `1`-by-`Life` vector of depreciation values with each element corresponding to a year of the asset's life.

Version History

Introduced before R2006a

See Also

`depfixdb` | `depgendb` | `deprdv` | `depstln`

Topics

“Analyzing and Computing Cash Flows” on page 2-11

depstln

Straight-line depreciation schedule

Syntax

```
Depreciation = depstln(Cost,Salvage,Life)
```

Description

`Depreciation = depstln(Cost,Salvage,Life)` computes the 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
```

Input Arguments

`Cost` — Initial value of the asset
numeric

Initial value of the asset, specified as a scalar numeric.

Data Types: `double`

`Salvage` — Salvage value of the asset
numeric

Salvage value of the asset, specified as a scalar numeric.

Data Types: `double`

`Life` — Depreciable life of the asset in years
numeric

Depreciable life of the asset in years, specified as a scalar numeric.

Data Types: `double`

Output Arguments

`Depreciation` — Depreciation

numeric

Depreciation, returned as a straight-line depreciation for an asset.

Version History

Introduced before R2006a

See Also

`depfixdb` | `depgendb` | `deprdv` | `depsoyd`

Topics

"Analyzing and Computing Cash Flows" on page 2-11

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 = [datetime(2000,11,6)
              datetime(2000,12,11)
              datetime(2001,1,15)
              datetime(2001,2,5)
              datetime(2001,3,4)
              datetime(2001,4,2)
              datetime(2001,4,30)
              datetime(2001,6,25)
              datetime(2001,9,4)
              datetime(2001,11,12)];
```

```
Settle = datetime(2000,11,3);
```

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 = 10×1

    0.0487
    0.0510
    0.0523
    0.0524
    0.0530
    0.0526
    0.0530
    0.0532
    0.0549
    0.0536


CurveDates = 10x1 datetime
   06-Nov-2000
   11-Dec-2000
   15-Jan-2001
   05-Feb-2001
   04-Mar-2001
   02-Apr-2001
   30-Apr-2001
   25-Jun-2001
   04-Sep-2001
   12-Nov-2001
```

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.

Input Arguments

`DiscRates` — Discount factors

decimal fraction

Discount factors, specified as a `NDATES`-by-`1` 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

datetime array | string array | date character vector

Maturity dates that correspond to the discount factors in `DiscRates`, specified as a `NDATES`-by-`1` column vector using a datetime array, string array, or date character vectors.

To support existing code, `disc2zero` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `datetime` | `string` | `char`

`Settle` — Common settlement date for discount rates in `DiscRates`

datetime scalar | string scalar | date character vector | serial date number

Common settlement date for the discount rates in `DiscRates`, specified as scalar datetime, string, or date character vector.

To support existing code, `disc2zero` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `datetime` | `string` | `char`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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 2-16.

Data Types: double

Output Arguments

ZeroRates — Zero curve for investment horizon represented by CurveDates

vector

Zero curve for the investment horizon represented by CurveDates, returned as a NDATES-by-1 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

vector

Maturity dates that correspond to the ZeroRates, returned as a NDATES-by-1 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.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although disc2zero supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)

y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

zero2disc | datetime | fwd2zero | prbyzero | pyld2zero | zbtprice | zbtyield | zero2disc | zero2fwd | zero2pyld

Topics

“Term Structure of Interest Rates” on page 2-29
“Fixed-Income Terminology” on page 2-15

discrate

Bank discount rate of security

Syntax

```
DiscRate = discrate(Settle,Maturity,Face,Price)
DiscRate = discrate(___,Basis)
```

Description

`DiscRate = discrate(Settle,Maturity,Face,Price)`computes 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.

`DiscRate = discrate(___,Basis)` adds an optional argument for `Basis`.

Examples

Compute the Bank Discount Rate of a Security

This example shows how to find the bank discount rate of a security.

```
DiscRate = discrate(datetime(2000,1,12),datetime(2000,6,25), 100, 97.74, 0)
```

```
DiscRate = 0.0501
```

Input Arguments

`Settle` — Settlement date
datetime scalar | string scalar | date character vector

Settlement date, specified as a scalar datetime, string, or date character vector.

To support existing code, `discrate` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `datetime` | `string` | `char`

`Maturity` — Maturity date
datetime scalar | string scalar | date character vector

Maturity date, specified as scalar datetime, string, or date character vector.

To support existing code, `discrate` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

Face — Redemption (par, face) value
numeric

Redemption (par, face) value, specified as a scalar numeric value.

Data Types: `double`

`Price` — Security price

numeric

Security price, specified as a scalar numeric value.

Data Types: `double`

`Basis` — Day-count basis of the instrument

`0` (actual/actual) (default) | integer with value of `0` to `13`

(Optional) Day-count basis of the instrument, specified as a scalar integer.

- 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 2-16.

Data Types: `double`

Output Arguments

`DiscRate` — Discount rate of security

decimal

Discount rate of security, returned as a decimal.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although `discrate` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Mayle. "*Standard Securities Calculation Methods.*" Volumes I-II, 3rd edition. Formula 2.

See Also

`acrudisc` | `fvdisc` | `prdisc` | `ylddisc` | `datetime`

Topics

"Term Structure of Interest Rates" on page 2-29
"Fixed-Income Terminology" on page 2-15

ecmlsrmle

Least-squares regression with missing data

Syntax

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

Description

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

$$Data_k \sim N(Design_k \times Parameters,\ Covariance)$$

for samples k = 1, ... , `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.

`[Parameters,Covariance,Resid,Info] = ecmlsrmle( ___ ,MaxIterations,TolParam, TolObj,Param0,Covar0,CovarFormat)` estimates a least-squares regression model with missing data using optional arguments.

Input Arguments

`Data` — Data sample

matrix

Data sample, specified as an `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 `mvnrmle` to handle missing data.)

Data Types: `double`

`Design` — Model design

matrix | cell array of character vectors

Model design, specified as 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.

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.

Data Types: double | cell

MaxIterations — Maximum number of iterations for the estimation algorithm

100 (default) | numeric

(Optional) Maximum number of iterations for the estimation algorithm, specified as a numeric. The default value is 100.

Data Types: double

TolParam — Convergence tolerance for estimation algorithm

sqrt(eps) (default) | numeric

(Optional) Convergence tolerance for estimation algorithm based on changes in model parameter estimates, specified as a numeric. The 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.

Data Types: double | table | timetable

TolObj — Convergence tolerance for estimation algorithm based on changes in objective function

eps ∧ 3/4 (default) | numeric

(Optional) Convergence tolerance for estimation algorithm based on changes in the objective function, specified as a numeric. The 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.

Data Types: double

Param0 — User-supplied initial estimate for the parameters of regression model

zero vector (default) | vector

(Optional) User-supplied initial estimate for the parameters of the regression model, specified as an NUMPARAMS-by-1 column vector.

Data Types: double

Covar0 — User-supplied initial or known estimate for the covariance matrix of the regression residuals

identity matrix (default) | matrix

(Optional) User-supplied initial or known estimate for the covariance matrix of the regression residuals, specified as an NUMSERIES-by-NUMSERIES 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.

Data Types: double

CovarFormat — Format for covariance matrix

'full' (default) | character vector with value 'full' or 'diagonal'

(Optional) Format for the covariance matrix, specified as a character vector. The choices are:

- 'full' — This is the default method that computes the full covariance matrix.
- 'diagonal' — This forces the covariance matrix to be a diagonal matrix.

Data Types: char

Output Arguments

Parameters — Parameters of the regression model

vector

Parameters of the regression model, returned as an NUMPARAMS-by-1 column vector of estimates for the parameters of the regression model.

Covariance — Covariance of the regression model's residuals

matrix

Covariance of the regression model's residuals, returned as an NUMSERIES-by-NUMSERIES matrix of estimates for the covariance of the regression model's residuals.

Resid — Residuals from the regression

matrix

Residuals from the regression, returned as an NUMSAMPLES-by-NUMSERIES matrix of residuals from the regression.

Info — Structure containing additional information from the regression

structure

Structure containing additional information from the regression, returned as a structure. 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 before the terminal iteration.
- `Info.PrevCovariance` – `NUMSERIES`-by-`NUMSERIES` matrix of estimates for the covariance parameters from the iteration just before the terminal iteration.

Use the estimates in the output structure `Info` for diagnostic purposes.

Version History

Introduced in R2006a

References

[1] 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.

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

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

[4] 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

`ecmlsrobj` | `ecmmvnrmle` | `ecmmvnrmle`

Topics

“Multivariate Normal Regression” on page 9-13
“Least-Squares Regression” on page 9-14
“Covariance-Weighted Least Squares” on page 9-14
“Feasible Generalized Least Squares” on page 9-15
“Seemingly Unrelated Regression” on page 9-16

ecmlsrobj

Log-likelihood function for least-squares regression with missing data

Syntax

```
Objective = ecmlsrobj(Data,Design,Parameters)
Objective = ecmlsrobj( ___ ,Covariance)
```

Description

`Objective = ecmlsrobj(Data,Design,Parameters)` 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.

`Objective = ecmlsrobj( ___ ,Covariance)` computes a least-squares objective function based on current parameter estimates with missing data using an optional argument.

Input Arguments

`Data` — Data sample

matrix

Data sample, specified as an `NUMSAMPLES`-by-`NUMSERIES` matrix with `NUMSAMPLES` samples of a `NUMSERIES`-dimensional random vector. If a data sample has missing values, represented as `NaN`s. Only samples that are entirely NaNs are ignored. (To ignore samples with at least one `NaN`, use `mvnrmle`.)

Data Types: `double`

`Design` — Model design

matrix | cell array of character vectors

Model design, specified as 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.

Data Types: `double` | `cell`

`Parameters` — Estimates for the parameters of regression model

vector

Estimates for the parameters of regression model, specified as an `NUMPARAMS`-by-`1` column vector.

Data Types: `double`

`Covariance` — User-supplied estimate for covariance matrix of residuals of the regression
vector

(Optional) User-supplied estimate for the covariance matrix of the residuals of the regression, specified as an `NUMPARAMS`-by-`1` column vector.

`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.

Data Types: `double`

Output Arguments

`Objective` — Least-squares objective function
scalar

Least-squares objective function, returned as scalar.

Version History

Introduced in R2006a

See Also

`ecmlsrmle` | `mvnrmle`

Topics

"Least-Squares Regression With Missing Data" on page 9-14
"Multivariate Normal Regression" on page 9-13
"Least-Squares Regression" on page 9-14
"Covariance-Weighted Least Squares" on page 9-14
"Feasible Generalized Least Squares" on page 9-15
"Seemingly Unrelated Regression" on page 9-16

ecmmvnrfish

Fisher information matrix for multivariate normal regression model

Syntax

```
Fisher] = ecmmvnrfish(Data,Design,Covariance)
Fisher = ecmmvnrfish( ___ ,Method,MatrixFormat,CovarFormat)
```

Description

`Fisher] = ecmmvnrfish(Data,Design,Covariance)` 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.

`Fisher = ecmmvnrfish( ___ ,Method,MatrixFormat,CovarFormat)` computes a Fisher information matrix based on current maximum likelihood or least-squares parameter estimates that account for missing data using optional arguments.

Input Arguments

Data — Data sample

matrix

Data sample, specified as an `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 `mvnrmle` to handle missing data.)

Data Types: `double`

Design — Model design

matrix | cell array of character vectors

Model design, specified as 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.

Data Types: `double` | `cell`

`Covariance` — Estimates for the covariance of the residuals of the regression
matrix

Estimates for the covariance of the residuals of the regression, specified as an `NUMSERIES`-by-`NUMSERIES` matrix.

Data Types: `double`

`Method` — Method of calculation for information matrix
`'hessian'` (default) | character vector with value `'hessian'` or `'fisher'`

(Optional) Method of calculation for the information matrix, specified as a character vector. The choices are:

- `'hessian'` — This is the default method using 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'` — This computes using the Fisher information matrix.

Data Types: `char`

`MatrixFormat` — Parameters to be included in Fisher information matrix
`'full'` (default) | character vector with value `'full'` or `'paramonly'`

(Optional) Parameters to be included in the Fisher information matrix, specified as a character vector. The choices are:

- `'full'` — This is the default method that computes the full Fisher information matrix for both model and covariance parameter estimates.
- `'paramonly'` — This computes only components of the Fisher information matrix associated with the model parameter estimates.

Data Types: `char`

`CovarFormat` — Format for covariance matrix
`'full'` (default) | character vector with value `'full'` or `'diagonal'`

(Optional) Format for the covariance matrix, specified as a character vector. The choices are:

- `'full'` — This is the default method that computes the full covariance matrix.
- `'diagonal'` — This forces the covariance matrix to be a diagonal matrix.

Data Types: `char`

Output Arguments

`Fisher` — Fisher information matrix

matrix

Fisher information matrix, returned as an `NUMPARAMS`-by-`NUMPARAMS` Fisher information matrix or Hessian matrix, depending on the optional input argument `Method`.

Version History

Introduced in R2006a

See Also

`ecmnmle` | `ecmnstd`

Topics

“Multivariate Normal Regression Without Missing Data” on page 9-13
“Multivariate Normal Regression” on page 9-13
“Least-Squares Regression” on page 9-14
“Covariance-Weighted Least Squares” on page 9-14
“Feasible Generalized Least Squares” on page 9-15
“Seemingly Unrelated Regression” on page 9-16
“Fisher Information” on page 9-4
“Multivariate Normal Linear Regression” on page 9-2

ecmmvnrmle

Multivariate normal regression with missing data

Syntax

```
[Param,Covar] = ecmmvnrmle(Data,Design)
[Param,Covar,Resid,Info] = ecmmvnrmle( ___ ,MaxIterations,TolParam,TolObj,
Param0,Covar0,CovarFormat)
```

Description

`[Param,Covar] = ecmmvnrmle(Data,Design)` estimates a multivariate normal regression model with missing data. The model has the form

$$Data_k \sim N(Design_k \times Parameters, Covariance)$$

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

`[Param,Covar,Resid,Info] = ecmmvnrmle( ___ ,MaxIterations,TolParam,TolObj, Param0,Covar0,CovarFormat)` adds an optional arguments for `MaxIterations`, `TolParam`, `TolObj`, `Param0`, `Covar0`, and `CovarFormat`.

Examples

Compute Multivariate Normal Regression With Missing Data

This example shows how to estimate a multivariate normal regression model with missing data.

First, load dates, total returns, and ticker symbols for the twelve stocks from the MAT-file.

```
load CAPMuniverse
whos Assets Data Dates
```

```
  Name           Size                Bytes  Class     Attributes

  Assets         1x14                 1568  cell                
  Data        1471x14               164752  double              
  Dates       1471x1                 11768  double              
```

```
Dates = datetime(Dates,'ConvertFrom','datenum');
```

The assets in the model have the following symbols, where the last two series are proxies for the market and the riskless asset.

```
Assets(1:14)
```

```
ans = 1x14 cell
    {'AAPL'}    {'AMZN'}    {'CSCO'}    {'DELL'}    {'EBAY'}    {'GOOG'}    {'HPQ'}    {'IBM'}
```

The data covers the period from January 1, 2000 to November 7, 2005 with daily total returns. Two stocks in this universe have missing values that are represented by NaNs. One of the two stocks had an IPO during this period and, consequently, has significantly less data than the other stocks.

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);

Alpha = NaN(1, length(NumAssets));
Beta = NaN(1, length(NumAssets));
Sigma = NaN(1, length(NumAssets));
StdAlpha = NaN(1, length(NumAssets));
StdBeta = NaN(1, length(NumAssets));
StdSigma = NaN(1, length(NumAssets));
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 the multivariate normal regression for each asset separately.
    [Param, Covar] = ecmmvnrmle(TestData, TestDesign)

end
```

```
Param = 2×1

    0.0012
    1.2294


Covar = 0.0010

Param = 2×1

    0.0006
    1.3661


Covar = 0.0020

Param = 2×1

   -0.0002
    1.5653


Covar = 8.8911e-04

Param = 2×1
```

```
   -0.0000
    1.2594


Covar = 6.4996e-04

Param = 2×1

    0.0014
    1.3441


Covar = 0.0014

Param = 2×1

    0.0046
    0.3742


Covar = 6.3272e-04

Param = 2×1

    0.0001
    1.3745


Covar = 6.5040e-04

Param = 2×1

   -0.0000
    1.0807


Covar = 2.8562e-04

Param = 2×1

    0.0001
    1.6002


Covar = 6.9146e-04

Param = 2×1

   -0.0002
    1.1765


Covar = 3.7138e-04

Param = 2×1

    0.0000
    1.5010


Covar = 0.0010
```

```
Param = 2×1

    0.0001
    1.6543


Covar = 0.0015
```

Input Arguments

Data — Data

matrix

Data, specified as an NUMSAMPLES-by-NUMSERIES matrix with NUMSAMPLES samples of a NUMSERIES-dimensional random vector. Missing values are indicated by NaNs. Only samples that are entirely NaNs are ignored. (To ignore samples with at least one NaN, use mvnrmle.)

Data Types: double

Design — Design model

matrix | cell array

Design model, specified as 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.

Data Types: double | cell

MaxIterations — Maximum number of iterations for the estimation algorithm

100 (default) | numeric

(Optional) Maximum number of iterations for the estimation algorithm, specified as a numeric.

Data Types: double

TolParam — Convergence tolerance for estimation algorithm based on changes in model parameter estimates

1.0e-8 (default) | numeric

(Optional) Convergence tolerance for estimation algorithm based on changes in model parameter estimates, specified as a numeric. 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, ... 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.

Data Types: `double`

`TolObj` — Convergence tolerance for estimation algorithm based on changes in objective function

`1.0e-12` (default) | numeric

(Optional) Convergence tolerance for estimation algorithm based on changes in the objective function, specified as a numeric. 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, ... 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.

Data Types: `double`

`Param0` — Estimate for the parameters of regression model

`[]` (default) | vector

(Optional) Estimate for the parameters of the regression model, specified as an `NUMPARAMS`-by-`1` column vector.

Data Types: `double`

`Covar0` — Estimate for the covariance matrix of regression residuals

`[]` (default) | matrix

(Optional) Estimate for the covariance matrix of the regression residuals, specified as `NUMSERIES`-by-`NUMSERIES` matrix.

Data Types: `double`

`CovarFormat` — Format for the covariance matrix

`'full'` (default) | character vector

(Optional) Format for the covariance matrix, specified as a character vector. The choices are:

- `'full'` — Compute the full covariance matrix.
- `'diagonal'` — Force the covariance matrix to be a diagonal matrix.

Data Types: `char`

Output Arguments

`Param` — Estimates for parameters of the regression model

vector

Estimates for the parameters of the regression model, returned as a `NUMPARAMS`-by-`1` column vector.

`Covar` — Estimates for the covariance of regression model's residuals

matrix

Estimates for the covariance of the regression model's residuals, returned as a `NUMSERIES`-by-`NUMSERIES` matrix.

Resid — Residuals from regression

matrix

Residuals from the regression, returned as a NUMSAMPLES-by-NUMSERIES matrix. 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.

Info — Additional information from regression

structure

Additional information from the regression, returned as a structure. 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.Info.PrevCovariance – NUMSERIES-by-NUMSERIES matrix of estimates for the covariance parameters from the iteration just prior to the terminal iteration.

Version History

Introduced in R2006a

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

ecmmvnrobj | mvnrmle

Topics

“Multivariate Normal Regression With Missing Data” on page 9-14
“Multivariate Normal Regression” on page 9-13
“Least-Squares Regression” on page 9-14
“Covariance-Weighted Least Squares” on page 9-14

“Feasible Generalized Least Squares” on page 9-15
“Seemingly Unrelated Regression” on page 9-16
“Multivariate Normal Linear Regression” on page 9-2

ecmmvnrobj

Log-likelihood function for multivariate normal regression with missing data

Syntax

```
Objective = ecmmvnrobj(Data,Design,Parameters,Covariance)
Objective = ecmmvnrobj( ___ ,CovarFormat)
```

Description

`Objective = ecmmvnrobj(Data,Design,Parameters,Covariance)` computes a log-likelihood function based on current maximum likelihood parameter estimates with missing data. `Objective` is a scalar that contains the least-squares objective function.

`Objective = ecmmvnrobj( ___ ,CovarFormat)` computes a log-likelihood function based on current maximum likelihood parameter estimates with missing data using an optional argument.

Input Arguments

`Data` — Data sample

matrix

Data sample, specified as an `NUMSAMPLES`-by-`NUMSERIES` matrix with `NUMSAMPLES` samples of a `NUMSERIES`-dimensional random vector. If a data sample has missing values, represented as `NaN`s. Only samples that are entirely NaNs are ignored. (To ignore samples with at least one `NaN`, use `mvnrmle`.)

Data Types: `double`

`Design` — Model design

matrix | cell array of character vectors

Model design, specified as 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.

Data Types: `double` | `cell`

`Parameters` — Estimates for the parameters of regression model

vector

Estimates for the parameters of regression model, specified as an `NUMPARAMS`-by-`1` column vector.

Data Types: `double`

`Covariance` — Estimates for covariance matrix of residuals of the regression
matrix

Estimates for the covariance matrix of the residuals of the regression, specified as an `NUMSERIES`-by-`NUMSERIES` matrix.

Data Types: `double`

`CovarFormat` — Format for covariance matrix
`'full'` (default) | character vector with value `'full'` or `'diagonal'`

(Optional) Format for the covariance matrix, specified as a character vector. The choices are:

- `'full'` — This is the default method that computes the full covariance matrix.
- `'diagonal'` — This forces the covariance matrix to be a diagonal matrix.

Data Types: `char`

Output Arguments

`Objective` — Least-squares objective function
scalar

Least-squares objective function, returned as scalar.

Version History

Introduced in R2006a

See Also

`ecmmvnrmle` | `mvnrmle` | `mvnrobj`

Topics
“Multivariate Normal Regression With Missing Data” on page 9-14
“Portfolios with Missing Data” on page 9-21
“Multivariate Normal Regression” on page 9-13
“Least-Squares Regression” on page 9-14
“Covariance-Weighted Least Squares” on page 9-14
“Feasible Generalized Least Squares” on page 9-15
“Seemingly Unrelated Regression” on page 9-16
“Multivariate Normal Linear Regression” on page 9-2

ecmmvnrstd

Evaluate standard errors for multivariate normal regression model

Syntax

```
[StdParameters,StdCovariance] = ecmmvnrstd(Data,Design,Covariance)
[StdParameters,StdCovariance] = ecmmvnrstd( ___ ,Method,CovarFormat)
```

Description

`[StdParameters,StdCovariance] = ecmmvnrstd(Data,Design,Covariance)` evaluates standard errors for a multivariate normal regression model with missing data. The model has the form

$$Data_k \sim N(Design_k \times Parameters, Covariance)$$

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

`[StdParameters,StdCovariance] = ecmmvnrstd( ___ ,Method,CovarFormat)` adds an optional arguments for `Method` and `CovarFormat`.

Examples

Compute Standard Errors for Multivariate Normal Regression

This example shows how to compute standard errors for a multivariate normal regression model.

First, load dates, total returns, and ticker symbols for the twelve stocks from the MAT-file.

```
load CAPMuniverse
whos Assets Data Dates
```

```
  Name            Size              Bytes  Class     Attributes

  Assets          1x14               1568  cell
  Data         1471x14             164752  double
  Dates        1471x1               11768  double
```

```
Dates = datetime(Dates,'ConvertFrom','datenum');
```

The assets in the model have the following symbols, where the last two series are proxies for the market and the riskless asset.

```
Assets(1:14)
```

```
ans = 1x14 cell
    {'AAPL'}    {'AMZN'}    {'CSCO'}    {'DELL'}    {'EBAY'}    {'GOOG'}    {'HPQ'}    {'IBM'}
```

The data covers the period from January 1, 2000 to November 7, 2005 with daily total returns. Two stocks in this universe have missing values that are represented by NaNs. One of the two stocks had an IPO during this period and, consequently, has significantly less data than the other stocks.

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

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);

Alpha = NaN(1, length(NumAssets));
Beta = NaN(1, length(NumAssets));
Sigma = NaN(1, length(NumAssets));
StdAlpha = NaN(1, length(NumAssets));
StdBeta = NaN(1, length(NumAssets));
StdSigma = NaN(1, length(NumAssets));
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);

    [Param, Covar] = ecmmvnrmle(TestData, TestDesign);

    % Estimate the sample standard errors for model parameters for each asset.
    StdParam = ecmmvnrstd(TestData, TestDesign, Covar,'hessian')

end
```

```
StdParam = 2×1

    0.0008
    0.0715


StdParam = 2×1

    0.0012
    0.1000


StdParam = 2×1

    0.0008
    0.0663


StdParam = 2×1

    0.0007
    0.0567


StdParam = 2×1

    0.0010
    0.0836


StdParam = 2×1

    0.0014
    0.2159


StdParam = 2×1

    0.0007
    0.0567


StdParam = 2×1

    0.0004
    0.0376


StdParam = 2×1

    0.0007
    0.0585


StdParam = 2×1

    0.0005
    0.0429


StdParam = 2×1

    0.0008
    0.0709


StdParam = 2×1

    0.0010
    0.0853
```

Input Arguments

Data — Data
matrix

Data, specified as an NUMSAMPLES-by-NUMSERIES matrix with NUMSAMPLES samples of a NUMSERIES-dimensional random vector. Missing values are indicated by NaNs. Only samples that are entirely NaNs are ignored. (To ignore samples with at least one NaN, use mvnrmle.)

Data Types: double

Design — Design model

matrix | cell array

Design model, specified as 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.

Data Types: double | cell

Covariance — Estimates for covariance of regression residuals

matrix

Estimates for the covariance of the regression residuals, specified as an NUMSERIES-by-NUMSERIES matrix.

Data Types: double

Method — Method of calculation for the information matrix

'hessian' (default) | character vector

(Optional) Method of calculation for the information matrix, specified as a character vector defined as:

- 'hessian' — 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' — The Fisher information matrix.

Note If Method = 'fisher', to obtain more quickly just the standard errors of variance estimates without the standard errors of the covariance estimates, set CovarFormat = 'diagonal' regardless of the form of the covariance matrix.

Data Types: char

CovarFormat — Format for the covariance matrix

'full' (default) | character vector

(Optional) Format for the covariance matrix, specified as a character vector. The choices are:

- 'full' — Compute the full covariance matrix.
- 'diagonal' — Force the covariance matrix to be a diagonal matrix.

Data Types: char

Output Arguments

StdParameters — Standard errors for each element of Parameters

vector

Standard errors for each element of Parameters, returned as an NUMPARAMS-by-1 column vector.

StdCovariance — Standard errors for each element of Covariance

matrix

Standard errors for each element of Covariance, returned as an NUMSERIES-by-NUMSERIES matrix.

Version History

Introduced in R2006a

References

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

See Also

ecmmvnrmle

Topics

"Multivariate Normal Regression" on page 9-13
"Least-Squares Regression" on page 9-14
"Covariance-Weighted Least Squares" on page 9-14
"Feasible Generalized Least Squares" on page 9-15
"Seemingly Unrelated Regression" on page 9-16
"Multivariate Normal Regression With Missing Data" on page 9-14
"Multivariate Normal Linear Regression" on page 9-2

ecmnfish

Fisher information matrix

Syntax

```
Fisher = ecmnfish(Data,Covariance)
Fisher = ecmnfish( ___ ,InvCovar,MatrixType)
```

Description

`Fisher = ecmnfish(Data,Covariance)` computes an NUMPARAMS-by-NUMPARAMS Fisher information matrix based on the current maximum likelihood parameter estimates.

Use `ecmnfish` after estimating the mean and covariance of `Data` with `ecmnmle`.

`Fisher = ecmnfish( ___ ,InvCovar,MatrixType)` adds optional arguments for `InvCovar` and `MatrixType`.

Examples

Compute Fisher Information Matrix Based on Parameter Estimates for Data

This example shows how to compute the Fisher information matrix based on paraemter estimates for `Data` for five years of daily total returns for 12 computer technology stocks, with six hardware and six software companies

```
load ecmtechdemo.mat
```

The time period for this 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.

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

```
ECMMean = 12×1

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

ECMCovar = 12×12
```

```
    0.0012    0.0005    0.0006    0.0005    0.0005    0.0003    0.0005    0.0003    0.0006    0.0
    0.0005    0.0024    0.0007    0.0006    0.0010    0.0004    0.0005    0.0003    0.0006    0.0
    0.0006    0.0007    0.0013    0.0007    0.0007    0.0003    0.0006    0.0004    0.0008    0.0
    0.0005    0.0006    0.0007    0.0009    0.0006    0.0002    0.0005    0.0003    0.0007    0.0
    0.0005    0.0010    0.0007    0.0006    0.0016    0.0006    0.0005    0.0003    0.0006    0.0
    0.0003    0.0004    0.0003    0.0002    0.0006    0.0022    0.0001    0.0002    0.0002    0.0
    0.0005    0.0005    0.0006    0.0005    0.0005    0.0001    0.0009    0.0003    0.0005    0.0
    0.0003    0.0003    0.0004    0.0003    0.0003    0.0002    0.0003    0.0005    0.0004    0.0
    0.0006    0.0006    0.0008    0.0007    0.0006    0.0002    0.0005    0.0004    0.0011    0.0
    0.0003    0.0004    0.0005    0.0004    0.0004    0.0001    0.0004    0.0003    0.0005    0.0
      ⋮
```

To evaluate the negative log-likelihood function for `ecmnmle`, use `ecmnfish` based on the current maximum likelihood parameter estimates for `ECMCovar`.

```
Fisher = ecmnfish(Data,ECMCovar)
```

```
Fisher = 90×90
10^7 ×

    0.0001    0.0000   -0.0000   -0.0000   -0.0000   -0.0000   -0.0000   -0.0000   -0.0000    0.0
    0.0000    0.0001   -0.0000    0.0000   -0.0000    0.0001    0.0000    0.0000    0.0000    0.0
   -0.0000   -0.0000    0.0002   -0.0000   -0.0000   -0.0000   -0.0000   -0.0000   -0.0000   -0.0
   -0.0000    0.0000   -0.0000    0.0003   -0.0000    0.0000   -0.0000   -0.0000   -0.0001   -0.0
   -0.0000   -0.0000   -0.0000   -0.0000    0.0001   -0.0000   -0.0000   -0.0000    0.0000   -0.0
   -0.0000    0.0001   -0.0000    0.0000   -0.0000    0.0002    0.0000   -0.0000    0.0000    0.0
   -0.0000    0.0000   -0.0000   -0.0000   -0.0000    0.0000    0.0002   -0.0001   -0.0000    0.0
   -0.0000    0.0000   -0.0000   -0.0000   -0.0000   -0.0000   -0.0001    0.0004   -0.0000   -0.0
   -0.0000    0.0000   -0.0000   -0.0001    0.0000    0.0000   -0.0000   -0.0000    0.0002   -0.0
    0.0000    0.0000   -0.0000   -0.0001   -0.0000    0.0001    0.0000   -0.0001   -0.0001    0.0
      ⋮
```

Input Arguments

`Data` — Data

matrix

Data, specified as an `NUMSAMPLES`-by-`NUMSERIES` matrix with `NUMSAMPLES` samples of a `NUMSERIES`-dimensional random vector. Missing values are indicated by `NaN`s.

Data Types: `double`

`Covariance` — Maximum likelihood parameter estimates for covariance of `Data`

matrix

Maximum likelihood parameter estimates for the covariance of the `Data` using the ECM algorithm, specified as a `NUMSERIES`-by-`NUMSERIES` matrix.

`InvCovar` — Cholesky decomposition of covariance matrix

`[ ]` (default) | matrix

(Optional) Inverse of covariance matrix, specified as a matrix using `inv` as:

```
inv(Covariance)
```

Data Types: `double`

`MatrixType` — Matrix format

`'full'` (default) | character vector

(Optional) Matrix format, specified as a character vector with a value of:

- `'full'` — Computes the full Fisher information matrix.
- `'meanonly'` — Computes only the components of the Fisher information matrix associated with the mean.

Data Types: `char`

Output Arguments

`Fisher` — Fisher information matrix

matrix

Fisher information matrix, returned as an `NUMPARAMSNUMPARAMS` matrix based on current parameter estimates, where `NUMPARAMS = NUMSERIES * (NUMSERIES + 3)/2` if the `MatrixFormat` = `'full'`. If the `MatrixFormat` = `'meanonly'`, then the `NUMPARAMS = NUMSERIES`.

Version History

Introduced before R2006a

See Also

`ecmnhess` | `ecmnmle`

Topics

"Multivariate Normal Regression With Missing Data" on page 9-14
"Fisher Information" on page 9-4

ecmnhess

Hessian of negative log-likelihood function

Syntax

```
Hessian = ecmnhess(Data,Covariance)
Hessian = ecmnhess( ___ ,InvCovar,MatrixType)
```

Description

`Hessian = ecmnhess(Data,Covariance)` computes an `NUMPARAMS`-by-`NUMPARAMS` Hessian matrix of the observed negative log-likelihood function based on current parameter estimates.

Use `ecmnhess` after estimating the mean and covariance of `Data` with `ecmnmle`.

`Hessian = ecmnhess( ___ ,InvCovar,MatrixType)` adds optional arguments for `InvCovar` and `MatrixType`.

Examples

Compute Hessian for Negative Log-Likelihood Function for Data

This example shows how to compute the Hessian for the negative log-likelihood function for five years of daily total return data for 12 computer technology stocks, with six hardware and six software companies

```
load ecmtechdemo.mat
```

The time period for this 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 `NaN`s. Also, Amazon (AMZN) had a few days with missing values scattered throughout the past five years.

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

```
ECMMean = 12×1

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

ECMCovar = 12×12
```

```
    0.0012    0.0005    0.0006    0.0005    0.0005    0.0003    0.0005    0.0003    0.0006    0.0
    0.0005    0.0024    0.0007    0.0006    0.0010    0.0004    0.0005    0.0003    0.0006    0.0
    0.0006    0.0007    0.0013    0.0007    0.0007    0.0003    0.0006    0.0004    0.0008    0.0
    0.0005    0.0006    0.0007    0.0009    0.0006    0.0002    0.0005    0.0003    0.0007    0.0
    0.0005    0.0010    0.0007    0.0006    0.0016    0.0006    0.0005    0.0003    0.0006    0.0
    0.0003    0.0004    0.0003    0.0002    0.0006    0.0022    0.0001    0.0002    0.0002    0.0
    0.0005    0.0005    0.0006    0.0005    0.0005    0.0001    0.0009    0.0003    0.0005    0.0
    0.0003    0.0003    0.0004    0.0003    0.0003    0.0002    0.0003    0.0005    0.0004    0.0
    0.0006    0.0006    0.0008    0.0007    0.0006    0.0002    0.0005    0.0004    0.0011    0.0
    0.0003    0.0004    0.0005    0.0004    0.0004    0.0001    0.0004    0.0003    0.0005    0.0
      ⋮
```

To evaluate the negative log-likelihood function for `ecmnmle`, use `ecmnhess` based on the current maximum likelihood parameter estimates for `ECMCovar`.

```
Hessian = ecmnhess(Data,ECMCovar)
```

```
Hessian = 90×90
10⁷ ×

    0.0001    0.0000   -0.0000   -0.0000   -0.0000   -0.0000   -0.0000   -0.0000   -0.0000    0.0
    0.0000    0.0001   -0.0000   -0.0000   -0.0000    0.0000   -0.0000    0.0000   -0.0000   -0.0
   -0.0000   -0.0000    0.0002   -0.0000   -0.0000   -0.0000   -0.0000   -0.0000   -0.0000   -0.0
   -0.0000   -0.0000   -0.0000    0.0003   -0.0000    0.0000   -0.0000   -0.0000   -0.0001   -0.0
   -0.0000   -0.0000   -0.0000   -0.0000    0.0001   -0.0000   -0.0000   -0.0000    0.0000   -0.0
   -0.0000    0.0000   -0.0000    0.0000   -0.0000    0.0000    0.0000   -0.0000    0.0000    0.0
   -0.0000   -0.0000   -0.0000   -0.0000   -0.0000    0.0000    0.0002   -0.0000   -0.0000   -0.0
   -0.0000    0.0000   -0.0000   -0.0000   -0.0000   -0.0000   -0.0000    0.0004   -0.0000   -0.0
   -0.0000   -0.0000   -0.0000   -0.0001    0.0000    0.0000   -0.0000   -0.0000    0.0002   -0.0
    0.0000   -0.0000   -0.0000   -0.0001   -0.0000    0.0000   -0.0000   -0.0000   -0.0001    0.0
      ⋮
```

Input Arguments

`Data` — Data

matrix

Data, specified as an `NUMSAMPLES`-by-`NUMSERIES` matrix with `NUMSAMPLES` samples of a `NUMSERIES`-dimensional random vector. Missing values are indicated by `NaN`s.

Data Types: `double`

`Covariance` — Maximum likelihood parameter estimates for covariance of `Data`

matrix

Maximum likelihood parameter estimates for the covariance of the `Data` using the ECM algorithm, specified as a `NUMSERIES`-by-`NUMSERIES` matrix.

`InvCovar` — Cholesky decomposition of covariance matrix

[ ] (default) | matrix

(Optional) Inverse of covariance matrix, specified as a matrix using `inv` as:

```
inv(Covariance)
```

Data Types: `double`

`MatrixType` — Matrix format

`'full'` (default) | character vector

(Optional) Matrix format, specified as a character vector with a value of:

- `'full'` — Computes the full Hessian matrix.
- `'meanonly'` — Computes only the components of the Hessian matrix associated with the mean.

Data Types: `char`

Output Arguments

`Hessian` — Hessian matrix

matrix

Hessian matrix, returned as an `NUMPARAMSNUMPARAMS` matrix of the observed log-likelihood function based on current parameter estimates, where `NUMPARAMS = NUMSERIES * (NUMSERIES + 3)/2` if the `MatrixFormat = 'full'`. If the `MatrixFormat = 'meanonly'`, then the `NUMPARAMS = NUMSERIES`.

Version History

Introduced before R2006a

See Also

`ecmnfish` | `ecmnmle`

Topics

"Maximum Likelihood Estimation" on page 9-3

ecmninit

Initial mean and covariance

Syntax

```
[Mean,Covariance] = ecmninit(Data,InitMethod)
[Mean,Covariance] = ecmninit(___,InitMethod)
```

Description

`[Mean,Covariance] = ecmninit(Data,InitMethod)` creates initial mean and covariance estimates for the function `ecmnmle`.

`[Mean,Covariance] = ecmninit(___,InitMethod)` adds an optional argument for `InitMethod`.

Examples

Compute Initial Mean and Covariance

This example shows how to compute the initial mean and covariance for five years of daily total return data for 12 computer technology stocks, with six hardware and six software companies

```
load ecmtechdemo.mat
```

The time period for this 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 `NaN`s. Also, Amazon (AMZN) had a few days with missing values scattered throughout the past five years.

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

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

```
NaNMean = 12×1

    0.0054
   -0.0006
   -0.0006
    0.0002
   -0.0009
    0.0042
    0.0011
   -0.0005
    0.0002
    0.0001
      ⋮
```

```
NaNCovar = 12×12
10^-3 ×

    0.7271    0.1003    0.0755    0.0585    0.1363    0.1030    0.0084    0.0741    0.0808    0.0
    0.1003    0.5958    0.1293    0.0919    0.2700    0.0554    0.0668    0.0548    0.1223    0.0
    0.0755    0.1293    0.2480    0.0841    0.0680    0.0322    0.0721    0.0632    0.1360    0.0
    0.0585    0.0919    0.0841    0.1414    0.0656   -0.0010    0.0386    0.0460    0.0617    0.0
    0.1363    0.2700    0.0680    0.0656    0.6223    0.2062    0.0797    0.0515    0.0850    0.0
    0.1030    0.0554    0.0322   -0.0010    0.2062    0.8376   -0.0103    0.0345    0.0236   -0.0
    0.0084    0.0668    0.0721    0.0386    0.0797   -0.0103    0.2462    0.0414    0.0881    0.0
    0.0741    0.0548    0.0632    0.0460    0.0515    0.0345    0.0414    0.1011    0.0561    0.0
    0.0808    0.1223    0.1360    0.0617    0.0850    0.0236    0.0881    0.0561    0.2642    0.0
    0.0407    0.0724    0.0562    0.0331    0.0436   -0.0034    0.0268    0.0321    0.0647    0.0
      ⋮
```

Input Arguments

Data — Data

matrix

Data, specified as an NUMSAMPLES-by-NUMSERIES matrix with NUMSAMPLES samples of a NUMSERIES-dimensional random vector. Missing values are indicated by NaNs.

Data Types: double

InitMethod — Initialization methods to compute initial estimates for mean and covariance of data

'nanskip' (default) | character vector

(Optional) Initialization methods to compute the initial estimates for the mean and covariance of data, specified as a character vector. The initialization methods are:

- 'nanskip' — Skip all records with NaNs.
- 'twostage' — Estimate mean. Fill NaNs with the mean. Then estimate the covariance.
- 'diagonal' — Form a diagonal covariance.

Data Types: char

Output Arguments

Mean — Initial estimate of mean of Data

vector

Initial estimate of the mean of the Data, returned as a NUMSERIES-by-1 column vector.

Covariance — Initial estimate of covariance of Data

matrix

Initial estimate of covariance of the Data, returned as a NUMSERIES-by-NUMSERIES matrix.

Algorithms

Model

The general model is

Z ~ *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.

Version History

Introduced before R2006a

See Also

`ecmnmle`

Topics

"Portfolios with Missing Data" on page 9-21
"Mean and Covariance Estimation" on page 9-4

ecmnmle

Mean and covariance of incomplete multivariate normal data

Syntax

```
ecmnmle(Data)
[Mean,Covariance] = ecmnmle(Data)
[Mean,Covariance] = ecmnmle(___,InitMethod,MaxIterations,Tolerance,Mean0,
Covar0)
```

Description

`ecmnmle(Data)` with no output arguments, this mode displays the convergence of the ECM algorithm in a plot by estimating objective function values for each iteration of the ECM algorithm until termination.

`[Mean,Covariance] = ecmnmle(Data)` estimates the mean and covariance of a data set (`Data`). 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 a conditional maximization form of the EM algorithm of Dempster, Laird, and Rubin [4].

`[Mean,Covariance] = ecmnmle(___,InitMethod,MaxIterations,Tolerance,Mean0, Covar0)` adds an optional arguments for `InitMethod`, `MaxIterations`, `Tolerance`,`Mean0`, and `Covar0`.

Examples

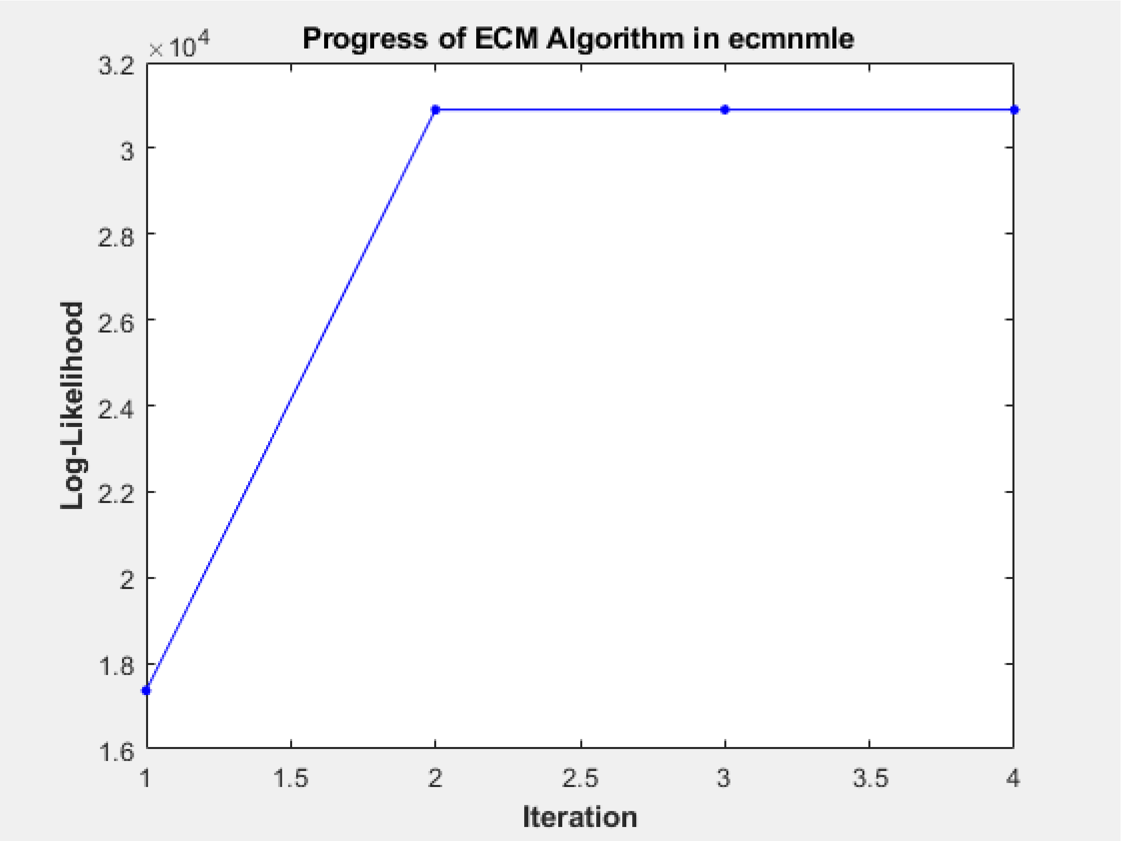
Compute Mean and Covariance of Incomplete Multivariate Normal Data

This example shows how to compute the mean and covariance of incomplete multivariate normal data for five years of daily total return data for 12 computer technology stocks, with six hardware and six software companies

```
load ecmtechdemo.mat
```

The time period for this 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 `NaN`s. Also, Amazon (AMZN) had a few days with missing values scattered throughout the past five years.

```
ecmnmle(Data)
```



```
ans = 12×1

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

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

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

```
Mean = 12×1

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

Covariance = 12×12

    0.0012    0.0005    0.0006    0.0005    0.0005    0.0003    0.0005    0.0003    0.0006    0.0
    0.0005    0.0024    0.0007    0.0006    0.0010    0.0004    0.0005    0.0003    0.0006    0.0
    0.0006    0.0007    0.0013    0.0007    0.0007    0.0003    0.0006    0.0004    0.0008    0.0
    0.0005    0.0006    0.0007    0.0009    0.0006    0.0002    0.0005    0.0003    0.0007    0.0
    0.0005    0.0010    0.0007    0.0006    0.0016    0.0006    0.0005    0.0003    0.0006    0.0
    0.0003    0.0004    0.0003    0.0002    0.0006    0.0022    0.0001    0.0002    0.0002    0.0
    0.0005    0.0005    0.0006    0.0005    0.0005    0.0001    0.0009    0.0003    0.0005    0.0
    0.0003    0.0003    0.0004    0.0003    0.0003    0.0002    0.0003    0.0005    0.0004    0.0
    0.0006    0.0006    0.0008    0.0007    0.0006    0.0002    0.0005    0.0004    0.0011    0.0
    0.0003    0.0004    0.0005    0.0004    0.0004    0.0001    0.0004    0.0003    0.0005    0.0
      ⋮
```

Input Arguments

`Data` — Data

matrix

Data, specified as an `NUMSAMPLES`-by-`NUMSERIES` matrix with `NUMSAMPLES` samples of a `NUMSERIES`-dimensional random vector. Missing values are indicated by `NaN`s.

Data Types: `double`

`InitMethod` — Initialization methods to compute initial estimates for mean and covariance of data

`'nanskip'` (default) | character vector

(Optional) Initialization methods to compute the initial estimates for the mean and covariance of data, specified as a character vector. The initialization methods are:

- `'nanskip'` — Skip all records with `NaN`s.
- `'twostage'` — Estimate mean. Fill `NaN`s with the mean. Then estimate the covariance.
- `'diagonal'` — Form a diagonal covariance.

Note If you supply `Mean0` and `Covar0`, `InitMethod` is not executed.

Data Types: `char`

`MaxIterations` — Maximum number of iterations

`50` (default) | numeric

(Optional) Maximum number of iterations for the expectation conditional maximization (ECM) algorithm, specified as a numeric.

Data Types: `double`

Tolerance — Convergence tolerance

1.0e-8 (default) | numeric

(Optional) Convergence tolerance for the ECM algorithm, specified as a numeric. If Tolerance ≤ 0, perform maximum iterations specified by MaxIterations and do not evaluate the objective function at each step unless in display mode.

Data Types: double

Mean0 — Estimate for the mean

[ ] (default) | matrix

(Optional) Estimate for the mean, specified as an NUMSERIES-by-1 column vector. If you leave Mean0 unspecified ([ ]), the method specified by InitMethod is used. If you specify Mean0, you must also specify Covar0.

Data Types: double

Covar0 — Estimate for the covariance

[ ] (default) | matrix

(Optional) Estimate for the covariance, specified as an NUMSERIES-by-NUMSERIES matrix, 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.

Data Types: double

Output Arguments

Mean — Maximum likelihood parameter estimates for mean of Data

vector

Maximum likelihood parameter estimates for the mean of the Data using ECM algorithm, returned as a NUMSERIES-by-1 column vector.

Covariance — Maximum likelihood parameter estimates for covariance of Data

matrix

Maximum likelihood parameter estimates for the covariance of the Data using ECM algorithm, returned as a NUMSERIES-by-NUMSERIES matrix.

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 NaNs and estimates initial values from complete-data records only. This initialization method tends to yield fastest convergence of the ECM algorithm. This routine switches to the twostage method if it determines that significant numbers of records contain NaN.
- twostage — The twostage method is the best choice for large problems (more than 10 series). It estimates the mean for each series using all available data for each series. It then estimates the covariance matrix with missing values treated as equal to the mean rather than as NaNs. This initialization method is 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.

Version History

Introduced before R2006a

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 9-14
“Portfolios with Missing Data” on page 9-21
“Mean and Covariance Estimation” on page 9-4

ecmnobj

Multivariate normal negative log-likelihood function

Syntax

```
Objective = ecmnobj(Data,Mean,Covariance)
Objective = ecmnobj(___,CholCovariance)
```

Description

`Objective = ecmnobj(Data,Mean,Covariance)` evaluates the negative log-likelihood function for `ecmnmle`.

Use `ecmnobj` after estimating the mean and covariance of `Data` with `ecmnmle`.

`Objective = ecmnobj(___,CholCovariance)` adds an optional argument for `CholCovariance`.

Examples

Compute Value of the Observed Negative Log-Likelihood Function for Data

This example shows how to compute the value of the observed negative log-likelihood function for five years of daily total return data for 12 computer technology stocks, with six hardware and six software companies

```
load ecmtechdemo.mat
```

The time period for this 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 `NaN`s. Also, Amazon (AMZN) had a few days with missing values scattered throughout the past five years.

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

```
ECMMean = 12×1

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


ECMCovar = 12×12
```

```
    0.0012    0.0005    0.0006    0.0005    0.0005    0.0003    0.0005    0.0003    0.0006    0.0
    0.0005    0.0024    0.0007    0.0006    0.0010    0.0004    0.0005    0.0003    0.0006    0.0
    0.0006    0.0007    0.0013    0.0007    0.0007    0.0003    0.0006    0.0004    0.0008    0.0
    0.0005    0.0006    0.0007    0.0009    0.0006    0.0002    0.0005    0.0003    0.0007    0.0
    0.0005    0.0010    0.0007    0.0006    0.0016    0.0006    0.0005    0.0003    0.0006    0.0
    0.0003    0.0004    0.0003    0.0002    0.0006    0.0022    0.0001    0.0002    0.0002    0.0
    0.0005    0.0005    0.0006    0.0005    0.0005    0.0001    0.0009    0.0003    0.0005    0.0
    0.0003    0.0003    0.0004    0.0003    0.0003    0.0002    0.0003    0.0005    0.0004    0.0
    0.0006    0.0006    0.0008    0.0007    0.0006    0.0002    0.0005    0.0004    0.0011    0.0
    0.0003    0.0004    0.0005    0.0004    0.0004    0.0001    0.0004    0.0003    0.0005    0.0
      ⋮
```

To evaluate the negative log-likelihood function for `ecmnmle`, use `ecmnobj` based on the current maximum likelihood parameter estimates.

```
Objective = ecmnobj(Data,ECMMean,ECMCovar)
```

```
Objective = -3.0898e+04
```

Input Arguments

`Data` — Data

matrix

Data, specified as an `NUMSAMPLES`-by-`NUMSERIES` matrix with `NUMSAMPLES` samples of a `NUMSERIES`-dimensional random vector. Missing values are indicated by `NaN`s.

Data Types: `double`

`Mean` — Maximum likelihood parameter estimates for mean of `Data`

vector

Maximum likelihood parameter estimates for the mean of the `Data` using the ECM algorithm, specified as a `NUMSERIES`-by-`1` column vector.

`Covariance` — Maximum likelihood parameter estimates for covariance of `Data`

matrix

Maximum likelihood parameter estimates for the covariance of the `Data` using the ECM algorithm, specified as a `NUMSERIES`-by-`NUMSERIES` matrix.

`CholCovariance` — Cholesky decomposition of covariance matrix

`[ ]` (default) | matrix

(Optional) Cholesky decomposition of covariance matrix, specified as a matrix using `chol` as:

```
chol(Covariance)
```

Data Types: `double`

Output Arguments

`Objective` — Value of the observed negative log-likelihood function over `Data`

numeric

Value of the observed negative log-likelihood function over the `Data`, returned as a numeric value.

Version History

Introduced before R2006a

See Also

`chol` | `ecmnmle`

Topics

“Multivariate Normal Regression Without Missing Data” on page 9-13
“Multivariate Normal Regression With Missing Data” on page 9-14

ecmnstd

Standard errors for mean and covariance of incomplete data

Syntax

```
[StdMean,StdCovar] = ecmnstd(Data,Mean,Covariance)
[StdMean,StdCovar] = ecmnstd( ___ ,Method)
```

Description

`[StdMean,StdCovar] = ecmnstd(Data,Mean,Covariance)` computes standard errors for mean and covariance of incomplete data.

Use `ecmnstd` 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).

`[StdMean,StdCovar] = ecmnstd( ___ ,Method)` adds an optional argument for `Method`.

Examples

Compute Standard Errors for Mean and Covariance of Incomplete Data

This example shows how to compute the standard errors for mean and covariance of incomplete data for five years of daily total return data for 12 computer technology stocks, with six hardware and six software companies

```
load ecmtechdemo.mat
```

The time period for this 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 `NaN`s. Also, Amazon (AMZN) had a few days with missing values scattered throughout the past five years.

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

```
ECMMean = 12×1

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

ECMCovar = 12×12

    0.0012    0.0005    0.0006    0.0005    0.0005    0.0003    0.0005    0.0003    0.0006    0.0
    0.0005    0.0024    0.0007    0.0006    0.0010    0.0004    0.0005    0.0003    0.0006    0.0
    0.0006    0.0007    0.0013    0.0007    0.0007    0.0003    0.0006    0.0004    0.0008    0.0
    0.0005    0.0006    0.0007    0.0009    0.0006    0.0002    0.0005    0.0003    0.0007    0.0
    0.0005    0.0010    0.0007    0.0006    0.0016    0.0006    0.0005    0.0003    0.0006    0.0
    0.0003    0.0004    0.0003    0.0002    0.0006    0.0022    0.0001    0.0002    0.0002    0.0
    0.0005    0.0005    0.0006    0.0005    0.0005    0.0001    0.0009    0.0003    0.0005    0.0
    0.0003    0.0003    0.0004    0.0003    0.0003    0.0002    0.0003    0.0005    0.0004    0.0
    0.0006    0.0006    0.0008    0.0007    0.0006    0.0002    0.0005    0.0004    0.0011    0.0
    0.0003    0.0004    0.0005    0.0004    0.0004    0.0001    0.0004    0.0003    0.0005    0.0
      ⋮
```

To evaluate the impact of the 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')
```

```
StdMeanF = 12×1

    0.0010
    0.0014
    0.0010
    0.0009
    0.0011
    0.0013
    0.0009
    0.0006
    0.0009
    0.0007
      ⋮
```

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')
```

```
StdMeanH = 12×1

    0.0010
    0.0014
    0.0010
    0.0009
    0.0011
    0.0021
    0.0009
    0.0006
    0.0009
    0.0007
      ⋮
```

The difference in the standard errors shows the increase in uncertainty of estimation of asset expected returns due to missing data. To view the differences:

```
Assets
```

```
Assets = 1x12 cell
    {'AAPL'}    {'AMZN'}    {'CSCO'}    {'DELL'}    {'EBAY'}    {'GOOG'}    {'HPQ'}    {'IBM'}
```

```
StdMeanH'
```

```
ans = 1×12

    0.0010    0.0014    0.0010    0.0009    0.0011    0.0021    0.0009    0.0006    0.0009    0.0
```

```
StdMeanF'
```

```
ans = 1×12

    0.0010    0.0014    0.0010    0.0009    0.0011    0.0013    0.0009    0.0006    0.0009    0.0
```

```
StdMeanH' - StdMeanF'
```

```
ans = 1×12
10^-3 ×

   -0.0000    0.0021   -0.0000   -0.0000   -0.0000    0.7742   -0.0000   -0.0000   -0.0000   -0.0
```

The two assets with missing data, AMZN and GOOG, are the only assets to have differences due to missing information.

Input Arguments

`Data` — Data

matrix

Data, specified as an `NUMSAMPLES`-by-`NUMSERIES` matrix with `NUMSAMPLES` samples of a `NUMSERIES`-dimensional random vector. Missing values are indicated by `NaN`s.

Data Types: `double`

`Mean` — Maximum likelihood parameter estimates for mean of `Data`

vector

Maximum likelihood parameter estimates for the mean of the `Data` using the ECM algorithm, specified as a `NUMSERIES`-by-`1` column vector.

Covariance — Maximum likelihood parameter estimates for covariance of Data
matrix

Maximum likelihood parameter estimates for the covariance of the Data using the ECM algorithm, specified as a NUMSERIES-by-NUMSERIES matrix.

Method — Method of estimation for standard error calculations
'hessian' (default) | character vector

(Optional) Method of estimation for standard error calculations, specified as a character vector. The estimation methods are:

- 'hessian' — The Hessian of the observed negative log-likelihood function. This method is recommended since the resultant standard errors incorporate the increase uncertainties due to missing data. In particular, standard errors calculated with the Hessian are generally larger than standard errors calculated with the Fisher information matrix.
- 'fisher' — The Fisher information matrix.

Data Types: char

Output Arguments

StdMean — Standard errors of estimates for each element of Mean vector
vector

Standard errors of estimates for each element of Mean vector, returned as a NUMSERIES-by-1 column vector.

StdCovar — Standard errors of estimates for each element of Covariance matrix
matrix

Standard errors of estimates for each element of Covariance matrix, returned as a NUMSERIES-by-NUMSERIES matrix.

Version History

Introduced before R2006a

See Also

ecmnmle

Topics

“Portfolios with Missing Data” on page 9-21
“Mean and Covariance Estimation” on page 9-4

effrr

Effective rate of return

Syntax

```
Return = effrr(Rate,NumPeriods)
```

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
```

Input Arguments

`Rate` — Annual percentage rate
scalar numeric decimal

Annual percentage rate, specified as a scalar numeric decimal.

Data Types: `double`

`NumPeriods` — Number of compounding periods per year
scalar integer

Number of compounding periods per year, specified as a scalar integer.

Data Types: `double`

Output Arguments

`Return` — Annual effective rate of return
scalar numeric decimal

Annual effective rate of return, returned as a scalar numeric decimal.

Version History

Introduced before R2006a

See Also

`nomrr`

Topics

"Analyzing and Computing Cash Flows" on page 2-11

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(MeanSigmaMAROrder)
```

Description

elpm(Mean,Sigma) compute expected lower partial moments (elpm) relative to a default value of MAR for each asset in a NUMORDERS-by-NUMSERIES matrix.

elpm(Mean,Sigma,MAR) computes expected lower partial moments (elpm) relative to a MAR for each asset in a NUMORDERS-by-NUMSERIES matrix.

elpm(Mean,Sigma,MAR,Order) computes expected lower partial moments (elpm) relative to a MAR and Order for each asset in a NUMORDERS-by-NUMSERIES matrix.

Moment = elpm(MeanSigmaMAROrder) computes expected lower partial moments (elpm) relative to a default value of MAR for each asset in a NUMORDERS-by-NUMSERIES matrix Moment.

Examples

Compute Expected Lower Partial Moments

This example shows how 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))
```

```
MAR = 0.0017
```

```
Mean = mean(Returns)
```

```
Mean = 1×3

    0.0038    0.0030    0.0017
```

```
Sigma = std(Returns, 1)
```

```
Sigma = 1×3

    0.0229    0.0389    0.0009
```

```
Assets
```

```
Assets = 1x3 cell
    {'Fund'}    {'Market'}    {'Cash'}
```

```
ELPM = elpm(Mean, Sigma, MAR, [0 1 2])
```

```
ELPM = 3×3

    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. The `elpm` 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)
```

```
RisklessCash = 0
```

Input Arguments

`Mean` — Mean returns

vector

Mean returns, specified as a `NUMSERIES` vector with mean returns for a collection of `NUMSERIES` assets.

Data Types: `double`

`Sigma` — Standard deviation of returns

vector

Standard deviation of returns, specified as a `NUMSERIES` vector with standard deviation of returns for a collection of `NUMSERIES` assets.

Data Types: `double`

`MAR` — Minimum acceptable return

`0` (default) | numeric

(Optional) Minimum acceptable return, specified as a scalar numeric. `MAR` is a cutoff level of return such that all returns above `MAR` contribute nothing to the lower partial moment.

Data Types: `double`

`Order` — Moment orders

`0` (default) | scalar numeric | vector

(Optional) Moment orders, specified as a either a scalar or a `NUMORDERS` vector of nonnegative integer moment orders. If no order specified, the default `Order = 0`, which is the shortfall probability. The `elpm` function does not work for negative or a noninteger `Order`.

Data Types: double

Output Arguments

Moment — Expected lower partial moments
matrix

Expected Lower partial moments, returned as 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. The output Moment for the lower partial moment represents the moments of asset returns that fall below a minimum acceptable level of return.

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.

More About

Lower Partial Moments

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.

Version History

Introduced in R2006b

References

[1] Bawa, V.S. "Safety-First, Stochastic Dominance, and Optimal Portfolio Choice." *Journal of Financial and Quantitative Analysis.* Vol. 13, No. 2, June 1978, pp. 255–271.

[2] Harlow, W.V. "Asset Allocation in a Downside-Risk Framework." *Financial Analysts Journal.* Vol. 47, No. 5, September/October 1991, pp. 28–40.

[3] Harlow, W.V. 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.

[4] Sortino, F.A. and Robert van der Meer. "Downside Risk." *Journal of Portfolio Management.* Vol. 17, No. 5, Spring 1991, pp. 27–31.

See Also

`lpm`

Topics

"Performance Metrics Overview" on page 7-2

emaxdrawdown

Compute expected maximum drawdown for Brownian motion

Syntax

```
ExpDrawdown = emaxdrawdown(Mu,Sigma,T)
```

Description

`ExpDrawdown = 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.

Examples

Compute Expected Maximum Drawdown

This example shows how to use log-return moments of a 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')
```

```
MaxDD = 0.1813
```

```
EMaxDD = emaxdrawdown(Mu, Sigma, T)
```

```
EMaxDD = 0.1545
```

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.

Input Arguments

Mu — Drift term of a Brownian motion with drift
numeric

Drift term of a Brownian motion with drift., specified as a scalar numeric.

Data Types: double

Sigma — Diffusion term of a Brownian motion with drift
numeric

Diffusion term of a Brownian motion with drift, specified as a scalar numeric.

Data Types: double

T — A time period of interest
numeric | vector

A time period of interest, specified as a scalar numeric or vector.

Data Types: double

Output Arguments

ExpDrawdown — Expected maximum drawdown
numeric

Expected maximum drawdown, returned as a numeric. 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 that emaxdrawdown supports.

Version History

Introduced in R2006b

References

[1] Malik, M. I., 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

“Performance Metrics Overview” on page 7-2

estimateCustomObjectivePortfolio

Estimate optimal portfolio for user-defined objective function for `Portfolio` object

Syntax

```
[pwgt,pbuy,psell,exitflag] = estimateCustomObjectivePortfolio(obj,fun)
[pwgt,pbuy,psell,exitflag] = estimateCustomObjectivePortfolio( ___ ,Name=Value)
```

Description

`[pwgt,pbuy,psell,exitflag] = estimateCustomObjectivePortfolio(obj,fun)` estimates optimal portfolio with a user-defined objective function for a `Portfolio` object. For details on using `estimateCustomObjectivePortfolio`, see “Solver Guidelines for Custom Objective Problems Using Portfolio Objects” on page 4-115.

`[pwgt,pbuy,psell,exitflag] = estimateCustomObjectivePortfolio( ___ ,Name=Value)` adds optional name-value arguments.

Examples

Solve Continuous Problems Using Custom Objectives

This example shows how to use `estimateCustomObjectivePortfolio` to solve a portfolio problem with a custom objective. You define the constraints for portfolio problems using functions for the `Portfolio` object and then you specify the objective function as an input to `estimateCustomObjectivePortfolio`. The objective function must be continuous (and preferably smooth).

Create `Portfolio` Object

Find the portfolio that solves the problem:

$$
\begin{aligned}
&\min_{x}\; x^T x \\
&s.t.\; \sum_{i} x_i = 1 \\
&\qquad x \geq 0
\end{aligned}
$$

Create a `Portfolio` object and set the default constraints using `setDefaultConstraints`. In this case, the portfolio problem does not involve the mean or covariance of the assets returns. Therefore, you do not need to define the assets moments to create the `Portfolio` object. You need only to define the number of assets in the problem.

```
% Create a Portfolio object
p = Portfolio(NumAssets=6);
% Define the constraints of the portfolio methods
p = setDefaultConstraints(p); % Long-only, fully invested weights
```

Define Objective Function

Define a function handle for the objective function $x^T x$.

```
% Define the objective function
objFun = @(x) x'*x;
```

Solve Portfolio Problem

Use `estimateCustomObjectivePortfolio` to compute the solution to the problem.

```
% Solve portfolio problem
wMin = estimateCustomObjectivePortfolio(p,objFun)
```

```
wMin = 6×1

    0.1667
    0.1667
    0.1667
    0.1667
    0.1667
    0.1667
```

The `estimateCustomObjectivePortfolio` function automatically assumes that the objective sense is to minimize. You can change that default behavior by using the `estimateCustomObjectivePortfolio` name-value argument `ObjectiveSense='maximize'`.

Add Gross or Net Return Constraints for Problems with Custom Objectives

This example shows how to use `estimateCustomObjectivePortfolio` to solve a portfolio problem with a custom objective and a return constraint. You define the constraints for portfolio problems, other than the return constraint, using functions for the `Portfolio` object and then you specify the objective function as an input to `estimateCustomObjectivePortfolio`. The objective function must be continuous (and preferably smooth). To specify a return constraint, you use the `TargetReturn` name-value argument.

Create `Portfolio` Object

Find the portfolio that solves the problem:

$$\begin{aligned} &\min_{x} \; x^T x \\ &s.t. \; \sum_i x_i = 1 \\ &\qquad x \geq 0 \end{aligned}$$

Create a `Portfolio` object and set the default constraints using `setDefaultConstraints`.

```
% Create a Portfolio object
load('SixStocks.mat')
p = Portfolio(AssetMean=AssetMean,AssetCovar=AssetCovar);
% Define the constraints of the portfolio methods
p = setDefaultConstraints(p); % Long-only, fully invested weights
```

Define Objective Function

Define a function handle for the objective function $x^T x$.

```
% Define the objective function
objFun = @(x) x'*x;
```

Add Return Constraints

When using the Portfolio object, you can use estimateFrontierByReturn to add a return constraint to the portfolio. However, when using the estimateCustomObjectivePortfolio function with a Portfolio object, you must add return constraints by using the TargetReturn name-value argument with a return target value.

The Portfolio object supports two types of return constraints: gross return and net return. The type of return constraint that is added to the portfolio problem is implicitly defined by whether you provide buy or sell costs to the Portfolio object using setCosts. If no buy or sell costs are present, the added return constraint is a gross return constraint. Otherwise, a net return constraint is added.

Gross Return Constraint

The gross portfolio return constraint for a portfolio is

$$r_0 + (\mu - r_0)^T x \geq \mu_0,$$

where r_0 is the risk-free rate (with 0 as default), μ is the mean of assets returns, and μ_0 is the target return.

Since buy or sell costs are not needed to add a gross return constraint, the Portfolio object does not need to be modified before using estimateCustomObjectivePortfolio.

```
% Set a return target
ret0 = 0.03;
% Solve portfolio problem with a gross return constraint
wGross = estimateCustomObjectivePortfolio(p,objFun, ...
    TargetReturn=ret0)
```

```
wGross = 6×1

    0.1377
    0.1106
    0.1691
    0.1829
    0.1179
    0.2818
```

The return constraint is not added to the Portfolio object. In other words, the Portfolio properties are not modified by adding a gross return constraint in estimateCustomObjectivePortfolio.

Net Return Constraint

The net portfolio return constraint for a portfolio is

$$r_0 + (\mu - r_0)^T x - c_B^T \max\{0, x - x_0\} - c_S^T \max\{0, x_0 - x\} \geq \mu_0,$$

where r_0 is the risk-free rate (with 0 as default), μ is the mean of assets returns, c_B is the proportional buy cost, c_S is the proportional sell cost, x_0 is the initial portfolio, and μ_0 is the target return.

To add net return constraints to the portfolio problem, you must use `setCosts` with the `Portfolio` object. If the `Portfolio` object has either of these costs, `estimateCustomObjectivePortfolio` automatically assumes that any added return constraint is a net return constraint.

```
% Add buy and sell costs to the Portfolio object
buyCost = 0.002;
sellCost = 0.001;
initPort = zeros(p.NumAssets,1);
p = setCosts(p,buyCost,sellCost,initPort);
% Solve portfolio problem with a net return constraint
% wNet = estimateCustomObjectivePortfolio(p,objFun, ...
%    TargetReturn=ret0)
```

As with the gross return constraint, the net return constraint is not added to the `Portfolio` object properties, however the `Portfolio` object is modified by the addition of buy or sell costs. Adding buy or sell costs to the `Portfolio` object does not affect any constraints besides the return constraint, but these costs do affect the solution of maximum return problems because the solution is the maximum *net* return instead of the maximum *gross* return.

Solve Portfolio Problems with Custom Objectives and Cardinality Constraints

This example shows how to use `estimateCustomObjectivePortfolio` to solve a portfolio problem with a custom objective and cardinality constraints. You define the constraints for portfolio problems using functions for the `Portfolio` object and then you specify the objective function as an input to `estimateCustomObjectivePortfolio`. For problems with cardinality constraints or continuous bounds, the objective function must be continuous and convex.

Create `Portfolio` Object

Find a long-only, fully weighted portfolio with half the assets that minimizes the tracking error to the equally weighted portfolio. Furthermore, if an asset is present in the portfolio, at least 10% should be invested in that asset. The porfolio problem is as follows:

$$\begin{aligned} &\min_x \ (x - x_0)^T \Sigma (x - x_0) \\ &s.t. \ \sum_i x_i = 1, \\ &\qquad \sum_i \#(x_i \neq 0) = 3, \\ &\qquad x_i \geq 0.1 \text{ or } x_i = 0 \end{aligned}$$

Create a `Portfolio` object and set the assets moments.

```
load('SixStocks.mat')
p = Portfolio(AssetMean=AssetMean,AssetCovar=AssetCovar);
nAssets = size(AssetMean,1);
```

Define the `Portfolio` constraints.

```
% Fully invested portfolio
p = setBudget(p,1,1);
% Cardinality constraint
p = setMinMaxNumAssets(p,3,3);
% Conditional bounds
p = setBounds(p,0.1,[],BoundType="conditional");
```

Define Objective Function

Define a function handle for the objective function.

```
% Define the objective function
EWP = 1/nAssets*ones(nAssets,1);
trackingError = @(x) (x-EWP)'*p.AssetCovar*(x-EWP);
```

Solve Portfolio Problem

Use `estimateCustomObjectivePortfolio` to compute the solution to the problem.

```
% Solve portfolio problem
wMinTE = estimateCustomObjectivePortfolio(p,trackingError)
```

```
wMinTE = 6×1

    0.1795
    0.3507
         0
    0.4698
         0
         0
```

Input Arguments

obj — Object for portfolio
`Portfolio` object

Object for portfolio, specified using a `Portfolio` object. When using a `Portfolio` object in the custom objective workflow, you do not need to use the mean-variance framework to estimate the mean and covariance. However, the custom objective workflow does require that you specify portfolio constraints. The `Portfolio` object that you use with the `estimateCustomObjectivePortfolio` function supports only the following constraints:

- Simple bounds — For more information, see “Working with 'Simple' Bound Constraints Using Portfolio Object” on page 4-61 and `setBounds`.
- Conditional bounds and cardinality constraints — For more information, see “Working with 'Conditional' BoundType, MinNumAssets, and MaxNumAssets Constraints Using Portfolio Objects” on page 4-78 and `setBounds` and `setMinMaxNumAssets`.

Note When using `estimateCustomObjectivePortfolio`, you cannot use tracking error constraints with conditional bounds and cardinality constraints. For more information, see “Solve Portfolio Problems with Custom Objectives and Cardinality Constraints” on page 15-852.

- Linear equalities — For more information, see “Working with Linear Equality Constraints Using Portfolio Object” on page 4-72 and `setEquality`.

- Linear inequalities — For more information, see “Working with Linear Inequality Constraints Using Portfolio Object” on page 4-75 and `setInequality`.
- Budget — For more information, see “Working with Budget Constraints Using Portfolio Object” on page 4-64 and `setBudget`.
- Group — For more information, see “Working with Group Constraints Using Portfolio Object” on page 4-66 and `setGroups`.
- Group ratio — For more information, see “Working with Group Ratio Constraints Using Portfolio Object” on page 4-69 and `setGroupRatio`.
- Turnover — For more information, see “Working with Average Turnover Constraints Using Portfolio Object” on page 4-81 and `setTurnover`.
- One-way turnover — For more information, see “Working with One-Way Turnover Constraints Using Portfolio Object” on page 4-84 and `setOneWayTurnover`.
- Tracking error — For more information, see “Working with Tracking Error Constraints Using Portfolio Object” on page 4-87 and `setTrackingError`.

Note If no initial portfolio is specified in `obj.InitPort`, the initial portfolio is assumed to be `0` so that `pbuy = max(0, pwgt)` and `psell = max(0, -pwgt)`. If no tracking portfolio is specified in `obj.TrackingPort`, the tracking portfolio is assumed to be `0`.

Data Types: `object`

`fun` — Function handle that defines objective function
function handle

Function handle that defines the objective function, specified using a function handle in terms of the portfolio weights.

Note The objective function must be continuous and defined using only the portfolio weights as variables. If the portfolio problem has cardinality constraints and/or conditional bounds using `setMinMaxNumAssets` or `setBounds`, the objective function must also be convex. For more information, see “Role of Convexity in Portfolio Problems” on page 4-148.

Data Types: `function_handle`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `pwgt = estimateCustomObjectivePortfolio(p,fun,ObjectiveSense="maximize",TargetReturn=0.05)`

`ObjectiveSense` — Sense of optimization
`"minimize"` (default) | string with value `"mimimize"` or `"maximize"` | character vector with value `'mimimize'` or `'maximize'`

Sense of the optimization, specified as `ObjectiveSense` and a string or character vector with one of the following values:

- `"minimize"` — The solution minimizes the objective function.
- `"maximize"` — The solution maximizes the objective function.

Data Types: `string` | `char`

`TargetReturn` — Target value for portfolio return
`[]` (default) | positive or negative scalar

Target value for the portfolio return, specified as `TargetReturn` and a positive or negative scalar.

Note In the `Portfolio` object, use `estimateFrontierByReturn` to add a return constraint to the portfolio. When you use `estimateCustomObjectivePortfolio`, return constraints are added by passing the name-value argument `TargetReturn` with the return target.

Data Types: `double`

Output Arguments

`pwgt` — Optimal weight allocation of portfolio problem
vector

Optimal weight allocation of the portfolio problem, returned as a `NumAssets`-by-`1` vector.

`pbuy` — Purchases relative to initial portfolio to achieve optimal weight allocation of portfolio problem
vector

Purchases relative to initial portfolio to achieve the optimal weight allocation of the portfolio problem, returned as a `NumAssets`-by-`1` vector.

`psell` — Sales relative to initial portfolio to achieve optimal weight allocation of the portfolio problem
vector

Sales relative to initial portfolio to achieve the optimal weight allocation of the portfolio problem, returned as a `NumAssets` vector.

`exitflag` — Reason solver stopped
enumeration variable | integer

Reason the solver stopped, returned as an enumeration variable or integer. There are two types of `exitflag` output. If the problem is continuous, the `exitflag` output is an enumeration variable. If the problem is mixed-integer, then the `exitflag` output is an integer.

Version History

Introduced in R2022b

References

[1] Cornuejols, G. and Reha Tütüncü. *Optimization Methods in Finance*. Cambridge University Press, 2007.

See Also

`estimatePortSharpeRatio` | `estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierByRisk`

Topics

“Diversify Portfolios Using Custom Objective” on page 4-329
“Portfolio Optimization Using Social Performance Measure” on page 4-257
“Portfolio Optimization Against a Benchmark” on page 4-195
“Solve Problem for Minimum Variance Portfolio with Tracking Error Penalty” on page 4-342
“Solve Problem for Minimum Tracking Error with Net Return Constraint” on page 4-347
“Solve Robust Portfolio Maximum Return Problem with Ellipsoidal Uncertainty” on page 4-349
“Risk Parity or Budgeting with Constraints” on page 4-356
“Single Period Goal-Based Wealth Management” on page 4-361
“Solver Guidelines for Custom Objective Problems Using Portfolio Objects” on page 4-115
“Portfolio Optimization Theory” on page 4-3
“Role of Convexity in Portfolio Problems” on page 4-148
“Troubleshooting estimateCustomObjectivePortfolio” on page 4-139
“solveContinuousCustomObjProb or solveMICustomObjProb Errors” on page 4-137

estimateAssetMoments

Estimate mean and covariance of asset returns from data

Note Using a `fints` object for the `AssetReturns` argument of `estimateAssetMoments` is not recommended. Use `timetable` instead for financial time series. For more information, see “Convert Financial Time Series Objects (fints) to Timetables”.

Syntax

```
obj = estimateAssetMoments(obj,AssetReturns)
obj = estimateAssetMoments( ___ ,Name,Value)
```

Description

`obj = estimateAssetMoments(obj,AssetReturns)` estimates mean and covariance of asset returns from data for a `Portfolio` object. For details on the workflow, see “Portfolio Object Workflow” on page 4-17.

`obj = estimateAssetMoments( ___ ,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` object 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 = 4×1

    0.0042
    0.0083
    0.0100
    0.0150


passetcovar = 4×4

    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 = 4×1

    0.0042
    0.0083
    0.0100
    0.0150


qassetcovar = 4×4

    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 = 4×1

    0.0042
    0.0083
    0.0100
    0.0150


passetcovar = 4×4

    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 = 4×1

    0.0042
    0.0083
    0.0100
    0.0150


qassetcovar = 4×4

    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
```

Estimate Mean and Covariance of Asset Returns from Timetable Data for a Portfolio Object

To illustrate using the `estimateAssetMoments` function with `AssetReturns` data continued in a `timetable` object, use the `CAPMuniverse.mat` which contains a `timetable` object (`AssetTimeTable`) for returns data.

```
load CAPMuniverse
AssetsTimeTable.Properties;
head(AssetsTimeTable,5)
```

```
        Time          AAPL        AMZN        CSCO        DELL        EBAY       GOOG      H
    ___________    _________    _________    _________    _________    _________    ____    ____

    03-Jan-2000     0.088805      0.1742     0.008775    -0.002353     0.12829     NaN       0.0
    04-Jan-2000    -0.084331    -0.08324     -0.05608     -0.08353   -0.093805     NaN     -0.07
    05-Jan-2000     0.014634    -0.14877    -0.003039     0.070984    0.066875     NaN     -0.00
    06-Jan-2000    -0.086538   -0.060072    -0.016619    -0.038847   -0.012302     NaN     -0.06
    07-Jan-2000     0.047368    0.061013       0.0587    -0.037708   -0.000964     NaN      0.02
```

Notice that GOOG has missing data (NaN), because it was not listed before Aug 2004. The estimateAssetMoments function has a name-value pair argument '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.

```
r = Portfolio;
r = estimateAssetMoments(r,AssetsTimeTable,'dataformat','returns','missingdata',true);
```

In addition, the estimateAssetMoments function also extracts asset names or identifiers from a timetable object when the name-value argument 'GetAssetList' set to true (its default value is false). If the 'GetAssetList' value is true, the timetable column identifiers are used to set the AssetList property of the Portfolio object. To show this, the formation of the Portfolio object r is repeated with the 'GetAssetList' flag set to true.

```
r = estimateAssetMoments(r,AssetsTimeTable,'GetAssetList',true);
disp(r.AssetList')
```

```
    {'AAPL'  }
    {'AMZN'  }
    {'CSCO'  }
    {'DELL'  }
    {'EBAY'  }
    {'GOOG'  }
    {'HPQ'   }
    {'IBM'   }
    {'INTC'  }
    {'MSFT'  }
    {'ORCL'  }
    {'YHOO'  }
    {'MARKET'}
    {'CASH'  }
```

Estimate Mean and Covariance of Asset Returns from Data for a Portfolio Object with Integrality Constraints

Create a Portfolio object for three assets.

```
AssetMean = [ 0.0101110; 0.0043532; 0.0137058 ];
AssetCovar = [ 0.00324625 0.00022983 0.00420395;
               0.00022983 0.00049937 0.00019247;
               0.00420395 0.00019247 0.00764097 ];
AssetMean = AssetMean/12
```

```
AssetMean = 3×1

    0.0008
    0.0004
    0.0011
```

```
AssetCovar = AssetCovar/12
```

```
AssetCovar = 3×3
10^-3 ×

    0.2705    0.0192    0.3503
    0.0192    0.0416    0.0160
    0.3503    0.0160    0.6367
```

```
X = portsim(AssetMean', AssetCovar, 120);

p = Portfolio('AssetMean',AssetMean, 'AssetCovar', AssetCovar);
p = setDefaultConstraints(p);
```

Use `setBounds` with semi-continuous constraints to set *xi*=0 or 0.02<=*xi*<=0.5 for all *i*=1,...NumAssets.

```
p = setBounds(p, 0.02, 0.5,'BoundType', 'Conditional', 'NumAssets', 3);
```

When working with a `Portfolio` object, the `setMinMaxNumAssets` function enables you to set up cardinality constraints for a long-only portfolio. This sets the cardinality constraints for the `Portfolio` object, where the total number of allocated assets satisfying the nonzero semi-continuous constraints are between `MinNumAssets` and `MaxNumAssets`. By setting `MinNumAssets=MaxNumAssets=2`, only two of the three assets are invested in the portfolio.

```
p = setMinMaxNumAssets(p, 2, 2);
```

Use `estimateAssetMoments` to estimate mean and covariance of asset returns from data for a `Portfolio` object.

```
p = estimateAssetMoments(p, X);
[passetmean, passetcovar] = getAssetMoments(p)
```

```
passetmean = 3×1

    0.0008
    0.0004
    0.0011
```

```
passetcovar = 3×3
10^-3 ×

    0.2705    0.0192    0.3503
    0.0192    0.0416    0.0160
    0.3503    0.0160    0.6367
```

The estimateAssetMoments function uses the MINLP solver to solve this problem. Use the setSolverMINLP function to configure the SolverType and options.

```
p.solverOptionsMINLP
```

```
ans = struct with fields:
                           MaxIterations: 1000
                    AbsoluteGapTolerance: 1.0000e-07
                    RelativeGapTolerance: 1.0000e-05
                  NonlinearScalingFactor: 1000
                  ObjectiveScalingFactor: 1000
                                 Display: 'off'
                           CutGeneration: 'basic'
                MaxIterationsInactiveCut: 30
                      ActiveCutTolerance: 1.0000e-07
                    IntMainSolverOptions: [1x1 optim.options.Intlinprog]
    NumIterationsEarlyIntegerConvergence: 30
```

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

Data Types: object

AssetReturns — Matrix, table, or timetable that contains asset price data that can be converted to asset returns

matrix | table | table

Matrix, table, or timetable that contains asset price data that can be converted to asset returns, specified by a NumSamples-by-NumAssets matrix.

AssetReturns data can be:

- NumSamples-by-NumAssets matrix.
- Table of NumSamples prices or returns at a given periodicity for a collection of NumAssets assets
- Timetable object with NumSamples observations and NumAssets time series

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 | table | timetable

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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 as the comma-separated pair consisting of `'DataFormat'` and 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 the comma-separated pair consisting of `'MissingData'` and 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 the comma-separated pair consisting of `'GetAssetList'` and 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 a table or timetable object.

If a `table` or `timetable` is passed into this function using the `AssetReturns` argument and the `GetAssetList` flag is `true`, the column names from the table or timetable 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);
```

Version History

Introduced in R2011a

See Also

`Portfolio` | `estimateBounds` | `portsim`

Topics

"Asset Returns and Moments of Asset Returns Using Portfolio Object" on page 4-41
"Portfolio Optimization Examples Using Financial Toolbox™" on page 4-152
"Portfolio Optimization Theory" on page 4-3

estimateBounds

Estimate global lower and upper bounds for set of portfolios

Syntax

```
[glb,gub,isbounded] = estimateBounds(obj)
[glb,gub,isbounded] = estimateBounds(obj,obtainExactBounds)
```

Description

`[glb,gub,isbounded] = estimateBounds(obj)` estimates global lower and upper bounds for set of portfolios for `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` objects. For details on the respective workflows when using these different objects, see “Portfolio Object Workflow” on page 4-17, “PortfolioCVaR Object Workflow” on page 5-16, and “PortfolioMAD Object Workflow” on page 6-15.

Note The `estimateBounds` function does not consider cardinality or semicontinuous constraints. For more information, see “Working with 'Conditional' BoundType, MinNumAssets, and MaxNumAssets Constraints Using Portfolio Objects” on page 4-78.

`[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 = 2×1

  -Inf
  -Inf


ub = 2×1

     0
   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 = 2×1

  -Inf
  -Inf


ub = 2×1

     0
   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 = 2×1

  -Inf
  -Inf


ub = 2×1

     0
   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.

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`

Data Types: `object`

`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.

Version History

Introduced in R2011a

See Also

`checkFeasibility`

Topics

"Validate the Portfolio Problem for Portfolio Object" on page 4-90
"Validate the CVaR Portfolio Problem" on page 5-78
"Validate the MAD Portfolio Problem" on page 6-76
"Portfolio Optimization Examples Using Financial Toolbox™" on page 4-152
"Portfolio Optimization Theory" on page 4-3

estimateFrontier

Estimate specified number of optimal portfolios on the efficient frontier

Syntax

```
[pwgt,pbuy,psell] = estimateFrontier(obj)
[pwgt,pbuy,psell] = estimateFrontier(obj,NumPorts)
```

Description

`[pwgt,pbuy,psell] = estimateFrontier(obj)` estimates the specified number of optimal portfolios on the efficient frontier for `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` objects. For details on the respective workflows when using these different objects, see “Portfolio Object Workflow” on page 4-17, “PortfolioCVaR Object Workflow” on page 5-16, and “PortfolioMAD Object Workflow” on page 6-15.

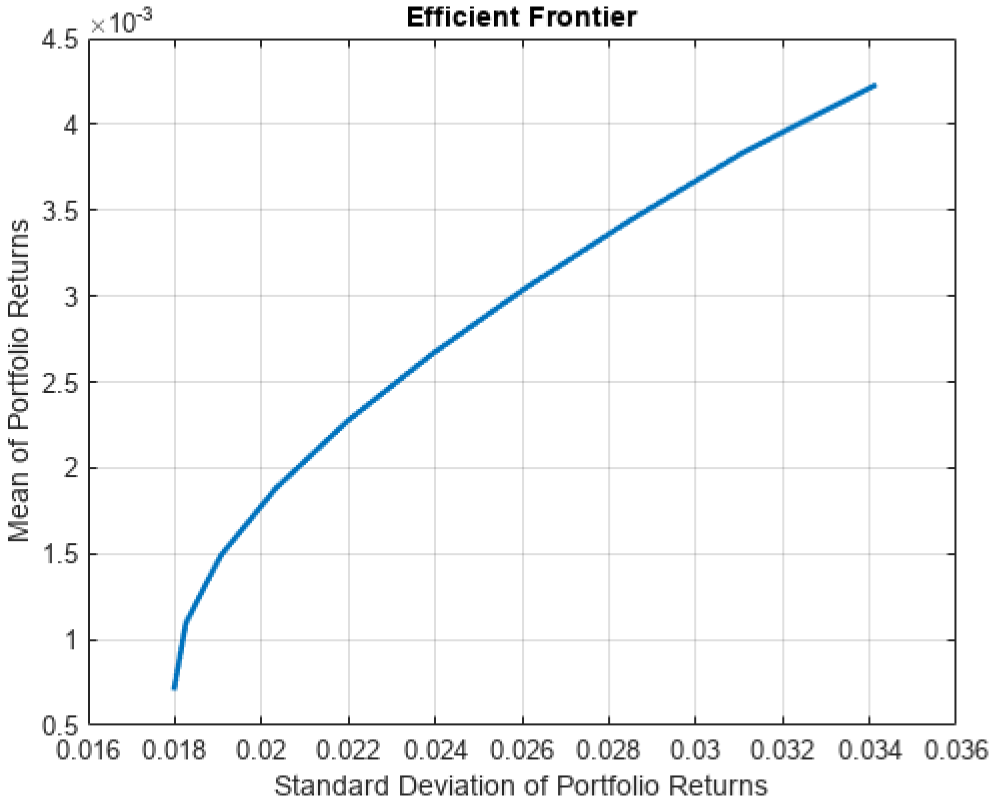
`[pwgt,pbuy,psell] = estimateFrontier(obj,NumPorts)` estimates the specified number of optimal portfolios on the efficient frontier with an additional option specified for `NumPorts`.

Examples

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
```

Create a Portfolio Object with BoundType and MaxNumAssets Constraints and Determine Efficient Portfolios

Create a Portfolio object for 12 stocks based on CAPMuniverse.mat.

```
load CAPMuniverse
p0 = Portfolio('AssetList',Assets(1:12));
p0 = estimateAssetMoments(p0, Data(:,1:12),'missingdata',true);
p0 = setDefaultConstraints(p0);
```

Use setMinMaxNumAssets to define a maximum number of 3 assets.

```
p1 = setMinMaxNumAssets(p0, [], 3);
```

Use setBounds to define a lower and upper bound and a BoundType of 'Conditional'.

```
p1 = setBounds(p1, 0.1, 0.5,'BoundType', 'Conditional');
pwgt = estimateFrontier(p1, 5);
```

The following table shows that the optimized allocations only have maximum 3 assets invested, and small positions less than 0.1 are avoided.

```
result = table(p0.AssetList', pwgt)
```

```
result=12×2 table
      Var1                                 pwgt                  
    ________    ___________________________________________________

    {'AAPL'}          0           0           0    0.14308           0
    {'AMZN'}          0           0           0          0           0
    {'CSCO'}          0           0           0          0           0
    {'DELL'}          0           0           0          0           0
    {'EBAY'}          0           0           0          0         0.5
    {'GOOG'}    0.16979     0.29587     0.42213    0.49998         0.5
    {'HPQ' }          0           0           0          0           0
    {'IBM' }    0.49602      0.4363     0.37309    0.35694           0
    {'INTC'}          0           0           0          0           0
    {'MSFT'}    0.33419     0.26783     0.20479          0           0
    {'ORCL'}          0           0           0          0           0
    {'YHOO'}          0           0           0          0           0
```

The estimateFrontier function uses the MINLP solver to solve this problem. Use the setSolverMINLP function to configure the SolverType and options.

```
p1.solverTypeMINLP
```

```
ans = 
'OuterApproximation'
```

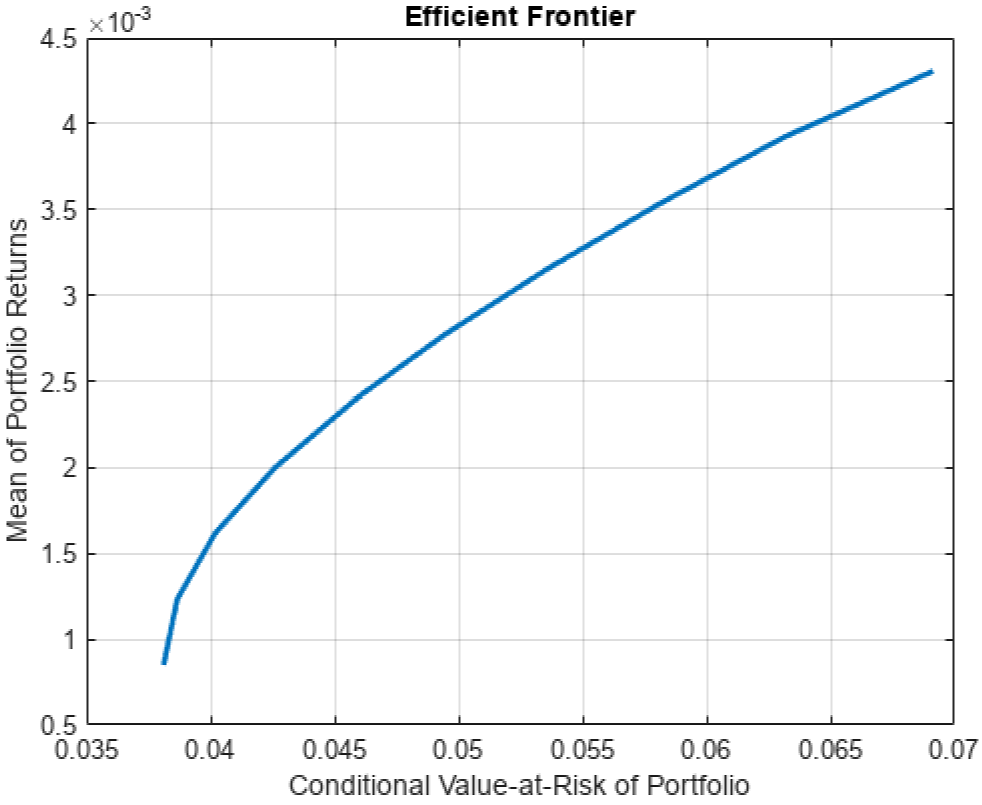
```
p1.solverOptionsMINLP
```

```
ans = struct with fields:
                          MaxIterations: 1000
                   AbsoluteGapTolerance: 1.0000e-07
                   RelativeGapTolerance: 1.0000e-05
                 NonlinearScalingFactor: 1000
                 ObjectiveScalingFactor: 1000
                                Display: 'off'
                          CutGeneration: 'basic'
               MaxIterationsInactiveCut: 30
                     ActiveCutTolerance: 1.0000e-07
                   IntMainSolverOptions: [1x1 optim.options.Intlinprog]
   NumIterationsEarlyIntegerConvergence: 30
```

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
```

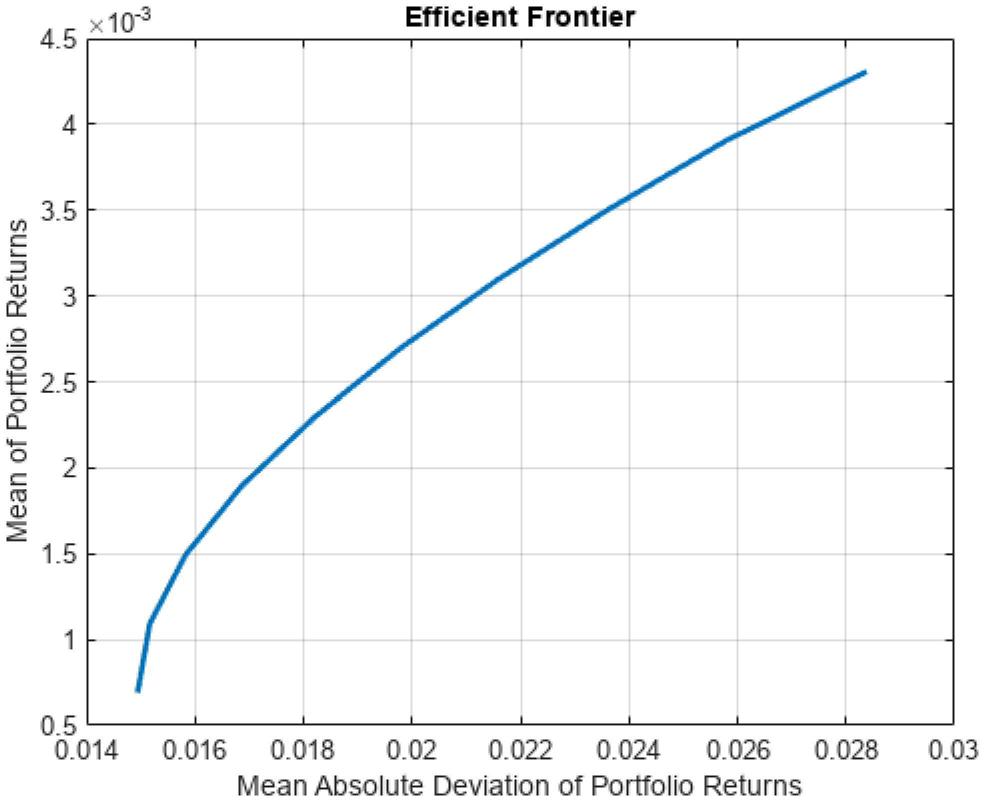
```
    pnames{i} = sprintf('Port%d',i);
end

Blotter = dataset([{pwgt},pnames],'obsnames',p.AssetList);

disp(Blotter);
```

```
            Port1       Port2        Port3       Port4      Port5
    AAPL    0.011002       0.07341     0.11855    0.12957    0
    AMZN           0             0           0          0    0
    CSCO           0             0           0          0    0
    DELL    0.023234             0           0          0    0
    EBAY           0             0           0          0    0
    GOOG     0.20304        0.3804     0.56259    0.75956    1
    HPQ     0.041781     0.0094108           0          0    0
    IBM       0.4452       0.36408      0.2625    0.11086    0
    INTC           0             0           0          0    0
    MSFT     0.27575        0.1727    0.056365          0    0
    ORCL           0             0           0          0    0
    YHOO           0             0           0          0    0
```

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    0.030785    0.074603     0.11383    0.13349    0
    AMZN           0           0           0          0    0
    CSCO           0           0           0          0    0
    DELL    0.010139           0           0          0    0
    EBAY           0           0           0          0    0
    GOOG      0.1607     0.35186     0.54435    0.74908    1
    HPQ     0.056834    0.024903           0          0    0
    IBM      0.45716     0.38008     0.29373    0.11743    0
    INTC           0           0           0          0    0
    MSFT     0.28438     0.16855    0.048097          0    0
    ORCL           0           0           0          0    0
    YHOO           0           0           0          0    0
```

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);
```

```
    0.8891    0.7215    0.5540    0.3865    0.2190    0.0515         0         0         0
    0.0369    0.1289    0.2209    0.3129    0.4049    0.4969    0.4049    0.2314    0.0579
    0.0404    0.0567    0.0730    0.0893    0.1056    0.1219    0.1320    0.1394    0.1468
    0.0336    0.0929    0.1521    0.2113    0.2705    0.3297    0.4630    0.6292    0.7953    1.0
```

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 = 4×10

    0.8891    0.7215    0.5540    0.3865    0.2190    0.0515         0         0         0
    0.0369    0.1289    0.2209    0.3129    0.4049    0.4969    0.4049    0.2314    0.0579
    0.0404    0.0567    0.0730    0.0893    0.1056    0.1219    0.1320    0.1394    0.1468
    0.0336    0.0929    0.1521    0.2113    0.2705    0.3297    0.4630    0.6292    0.7953    1.0
```

```
display(pbuy);
```

```
pbuy = 4×10

    0.5891    0.4215    0.2540    0.0865         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    0.0521    0.1113    0.1705    0.2297    0.3630    0.5292    0.6953    0.9
```

```
display(psell);
```

```
psell = 4×10

         0         0         0         0    0.0810    0.2485    0.3000    0.3000    0.3000    0.3
    0.2631    0.1711    0.0791         0         0         0         0    0.0686    0.2421    0.3
    0.1596    0.1433    0.1270    0.1107    0.0944    0.0781    0.0680    0.0606    0.0532    0.2
    0.0664    0.0071         0         0         0         0         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);
```

```
    0.8454    0.6847    0.5166    0.3541    0.1897    0.0315         0         0         0
    0.0606    0.1429    0.2281    0.3167    0.3989    0.4732    0.3531    0.1804         0
    0.0456    0.0638    0.0944    0.1079    0.1344    0.1583    0.1733    0.1919    0.2212
    0.0484    0.1085    0.1609    0.2213    0.2770    0.3370    0.4736    0.6277    0.7788    1.0
```

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 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 = 4×10

    0.8454    0.6847    0.5166    0.3541    0.1897    0.0315         0         0         0
    0.0606    0.1429    0.2281    0.3167    0.3989    0.4732    0.3531    0.1804         0
    0.0456    0.0638    0.0944    0.1079    0.1344    0.1583    0.1733    0.1919    0.2212
    0.0484    0.1085    0.1609    0.2213    0.2770    0.3370    0.4736    0.6277    0.7788    1.0
```

```
display(pbuy);
```

```
pbuy = 4×10

    0.5454    0.3847    0.2166    0.0541         0         0         0         0         0
         0         0         0    0.0167    0.0989    0.1732    0.0531         0         0
         0         0         0         0         0         0         0         0    0.0212
         0    0.0085    0.0609    0.1213    0.1770    0.2370    0.3736    0.5277    0.6788    0.9
```

```
display(psell);
```

```
psell = 4×10

         0         0         0         0    0.1103    0.2685    0.3000    0.3000    0.3000    0.3
    0.2394    0.1571    0.0719         0         0         0         0    0.1196    0.3000    0.3
    0.1544    0.1362    0.1056    0.0921    0.0656    0.0417    0.0267    0.0081         0    0.2
    0.0516         0         0         0         0         0         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);
```

```
    0.8823    0.7151    0.5490    0.3819    0.2175    0.0499         0         0         0
    0.0420    0.1290    0.2130    0.2971    0.3822    0.4667    0.3615    0.1752         0
    0.0394    0.0600    0.0822    0.1068    0.1238    0.1487    0.1780    0.2101    0.2267
    0.0363    0.0959    0.1557    0.2142    0.2765    0.3347    0.4605    0.6147    0.7733    1.0
```

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 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 = 4×10

    0.8823    0.7151    0.5490    0.3819    0.2175    0.0499         0         0         0
    0.0420    0.1290    0.2130    0.2971    0.3822    0.4667    0.3615    0.1752         0
    0.0394    0.0600    0.0822    0.1068    0.1238    0.1487    0.1780    0.2101    0.2267
    0.0363    0.0959    0.1557    0.2142    0.2765    0.3347    0.4605    0.6147    0.7733    1.0
```

```
display(pbuy);
```

```
pbuy = 4×10

    0.5823    0.4151    0.2490    0.0819         0         0         0         0         0
         0         0         0         0    0.0822    0.1667    0.0615         0         0
         0         0         0         0         0         0         0    0.0101    0.0267
         0         0    0.0557    0.1142    0.1765    0.2347    0.3605    0.5147    0.6733    0.9
```

```
display(psell);
```

```
psell = 4×10

         0         0         0         0    0.0825    0.2501    0.3000    0.3000    0.3000    0.3
    0.2580    0.1710    0.0870    0.0029         0         0         0    0.1248    0.3000    0.3
    0.1606    0.1400    0.1178    0.0932    0.0762    0.0513    0.0220         0         0    0.2
    0.0637    0.0041         0         0         0         0         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.

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`

Data Types: `object`

`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 15-881.

Version History

Introduced in R2011a

References

[1] Cornuejols, G., and R. Tutuncu. *Optimization Methods in Finance.* Cambridge University Press, 2007.

See Also

`estimateFrontierByReturn` | `estimateFrontierByRisk` | `estimateFrontierLimits` | `setBounds` | `setMinMaxNumAssets`

Topics

"Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object" on page 4-94
"Estimate Efficient Frontiers for Portfolio Object" on page 4-118
"Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object" on page 5-82
"Estimate Efficient Frontiers for PortfolioCVaR Object" on page 5-102
"Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object" on page 6-80
"Estimate Efficient Frontiers for PortfolioMAD Object" on page 6-97
"Portfolio Optimization Examples Using Financial Toolbox™" on page 4-152
"Bond Portfolio Optimization Using Portfolio Object" on page 10-30
"Portfolio Optimization Theory" on page 4-3

estimateFrontierByReturn

Estimate optimal portfolios with targeted portfolio returns

Syntax

```
[pwgt,pbuy,psell] = estimateFrontierByReturn(obj,TargetReturn)
```

Description

`[pwgt,pbuy,psell] = estimateFrontierByReturn(obj,TargetReturn)` estimates optimal portfolios with targeted portfolio returns for `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` objects. For details on the respective workflows when using these different objects, see “Portfolio Object Workflow” on page 4-17, “PortfolioCVaR Object Workflow” on page 5-16, and “PortfolioMAD Object Workflow” on page 6-15.

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 = 4×3

    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 Portfolios with Targeted Portfolio Returns for a Portfolio Object with BoundType, MinNumAsset, and MaxNumAsset Constraints

When any one, or any combination of the constraints from 'Conditional' BoundType, MinNumAssets, and MaxNumAssets are active, the portfolio problem is formulated as mixed integer programming problem and the MINLP solver is used.

Create a Portfolio object for three assets.

```
AssetMean = [ 0.0101110; 0.0043532; 0.0137058 ];
AssetCovar = [ 0.00324625 0.00022983 0.00420395;
               0.00022983 0.00049937 0.00019247;
               0.00420395 0.00019247 0.00764097 ];
p = Portfolio('AssetMean', AssetMean, 'AssetCovar', AssetCovar);
p = setDefaultConstraints(p);
```

Use setBounds with semicontinuous constraints to set *xi* = 0 or 0.02 <= *xi* <= 0.5 for all *i* = 1,...NumAssets.

```
p = setBounds(p, 0.02, 0.7,'BoundType', 'Conditional', 'NumAssets', 3);
```

When working with a Portfolio object, the setMinMaxNumAssets function enables you to set up the limits on the number of assets invested (as known as cardinality) constraints. This sets the total number of allocated assets satisfying the Bound constraints that are between MinNumAssets and MaxNumAssets. By setting MinNumAssets = MaxNumAssets = 2, only two of the three assets are invested in the portfolio.

```
p = setMinMaxNumAssets(p, 2, 2);
```

Use estimateFrontierByReturn to estimate optimal portfolios with targeted portfolio returns.

```
[pwgt, pbuy, psell] = estimateFrontierByReturn(p,[ 0.0072321, 0.0119084 ])
```

```
pwgt = 3×2

         0    0.5000
    0.6922         0
    0.3078    0.5000


pbuy = 3×2

         0    0.5000
    0.6922         0
    0.3078    0.5000


psell = 3×2

     0     0
     0     0
     0     0
```

The estimateFrontierByReturn function uses the MINLP solver to solve this problem. Use the setSolverMINLP function to configure the SolverType and options.

```
p.solverTypeMINLP
```

```
ans =
'OuterApproximation'
```

```
p.solverOptionsMINLP
```

```
ans = struct with fields:
                         MaxIterations: 1000
                  AbsoluteGapTolerance: 1.0000e-07
                  RelativeGapTolerance: 1.0000e-05
                NonlinearScalingFactor: 1000
                ObjectiveScalingFactor: 1000
                               Display: 'off'
                         CutGeneration: 'basic'
              MaxIterationsInactiveCut: 30
                    ActiveCutTolerance: 1.0000e-07
                  IntMainSolverOptions: [1x1 optim.options.Intlinprog]
    NumIterationsEarlyIntegerConvergence: 30
```

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 = 4×3

    0.7371    0.3072         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 = 4×3

    0.7437    0.3146         0
    0.1356    0.3837    0.4423
    0.0326    0.0939    0.1323
    0.0881    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.

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`

Data Types: `object`

`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);
```

Version History

Introduced in R2011a

See Also

`estimateFrontier` | `estimateFrontierByRisk` | `estimateFrontierLimits` | `setBounds` | `setMinMaxNumAssets`

Topics

“Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-94
“Estimate Efficient Frontiers for Portfolio Object” on page 4-118
“Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-82
“Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-102
“Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object” on page 6-80
“Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-97
“Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
“Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215
“Portfolio Optimization Using Factor Models” on page 4-224
“Portfolio Optimization Theory” on page 4-3

estimateFrontierByRisk

Estimate optimal portfolios with targeted portfolio risks

Syntax

`[pwgt,pbuy,psell] = estimateFrontierByRisk(obj,TargetRisk)`

`[pwgt,pbuy,psell] = estimateFrontierByRisk( ___ ,Name,Value)`

Description

`[pwgt,pbuy,psell] = estimateFrontierByRisk(obj,TargetRisk)` estimates optimal portfolios with targeted portfolio risks for `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` objects. For details on the respective workflows when using these different objects, see “Portfolio Object Workflow” on page 4-17, “PortfolioCVaR Object Workflow” on page 5-16, and “PortfolioMAD Object Workflow” on page 6-15.

`[pwgt,pbuy,psell] = estimateFrontierByRisk( ___ ,Name,Value)` adds name-optional name-value pair arguments for `Portfolio` or `PortfolioMAD` objects.

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%. This example uses the default `'direct'` method to estimate the optimal portfolios with targeted portfolio risks.

```
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 = 4×3

    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 `Portfolio` Object with `BoundType`, `MinNumAsset`, and `MaxNumAsset` Constraints

When any one, or any combination of the constraints from `'Conditional'` `BoundType`, `MinNumAssets`, and `MaxNumAssets` are active, the portfolio problem is formulated as mixed integer programming problem and the MINLP solver is used.

Create a `Portfolio` object for three assets.

```
AssetMean = [ 0.0101110; 0.0043532; 0.0137058 ];
AssetCovar = [ 0.00324625 0.00022983 0.00420395;
               0.00022983 0.00049937 0.00019247;
               0.00420395 0.00019247 0.00764097 ];
p = Portfolio('AssetMean', AssetMean, 'AssetCovar', AssetCovar);
p = setDefaultConstraints(p);
```

Use `setBounds` with semicontinuous constraints to set *xi* = `0` or `0.02` <= *`xi`* <= `0.5` for all *i* = `1`,...`NumAssets`.

```
p = setBounds(p, 0.02, 0.7,'BoundType', 'Conditional', 'NumAssets', 3);
```

When working with a `Portfolio` object, the `setMinMaxNumAssets` function enables you to set up the limits on the number of assets invested (as known as cardinality) constraints. This sets the total number of allocated assets satisfying the Bound constraints that are between `MinNumAssets` and `MaxNumAssets`. By setting `MinNumAssets` = `MaxNumAssets` = 2, only two of the three assets are invested in the portfolio.

```
p = setMinMaxNumAssets(p, 2, 2);
```

Use `estimateFrontierByRisk` to estimate optimal portfolios with targeted portfolio risks.

```
[pwgt, pbuy, psell] = estimateFrontierByRisk(p,[0.0324241, 0.0694534 ])
```

```
pwgt = 3×2

    0.0000    0.5000
    0.6907    0.0000
    0.3093    0.5000


pbuy = 3×2

    0.0000    0.5000
    0.6907    0.0000
    0.3093    0.5000


psell = 3×2

     0     0
     0     0
     0     0
```

The `estimateFrontierByRisk` function uses the MINLP solver to solve this problem. Use the `setSolverMINLP` function to configure the `SolverType` and options.

```
p.solverTypeMINLP

ans = 
'OuterApproximation'

p.solverOptionsMINLP

ans = struct with fields:
                           MaxIterations: 1000
                    AbsoluteGapTolerance: 1.0000e-07
                    RelativeGapTolerance: 1.0000e-05
                  NonlinearScalingFactor: 1000
                  ObjectiveScalingFactor: 1000
                                 Display: 'off'
                           CutGeneration: 'basic'
                MaxIterationsInactiveCut: 30
                      ActiveCutTolerance: 1.0000e-07
                    IntMainSolverOptions: [1x1 optim.options.Intlinprog]
    NumIterationsEarlyIntegerConvergence: 30
```

Obtain Portfolios with Targeted Portfolio Risks for a Portfolio Object Using the Direct Method and Solver Options

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%. This example uses the default`'direct'` method to estimate the optimal portfolios with targeted portfolio risks. The `'direct'` method uses `fmincon` to solve the optimization problem that maximizes portfolio return, subject to the target risk as the quadratic nonlinear constraint. `setSolver` specifies the `solverType` and `SolverOptions` for `fmincon`.

```
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);

 p = setSolver(p, 'fmincon', 'Display', 'off', 'Algorithm', 'sqp', ...
        'SpecifyObjectiveGradient', true, 'SpecifyConstraintGradient', true, ...
        'ConstraintTolerance', 1.0e-8, 'OptimalityTolerance', 1.0e-8, 'StepTolerance', 1.0e-8);

 pwgt = estimateFrontierByRisk(p, [0.12, 0.14, 0.16]);

 display(pwgt);

pwgt = 4×3

    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 = 4×3

    0.5363    0.1387         0
    0.2655    0.4991    0.3830
    0.0570    0.1239    0.1461
    0.1412    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%. This example uses the default `'direct'` method to estimate the optimal portfolios with targeted portfolio risks.

```
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);
```

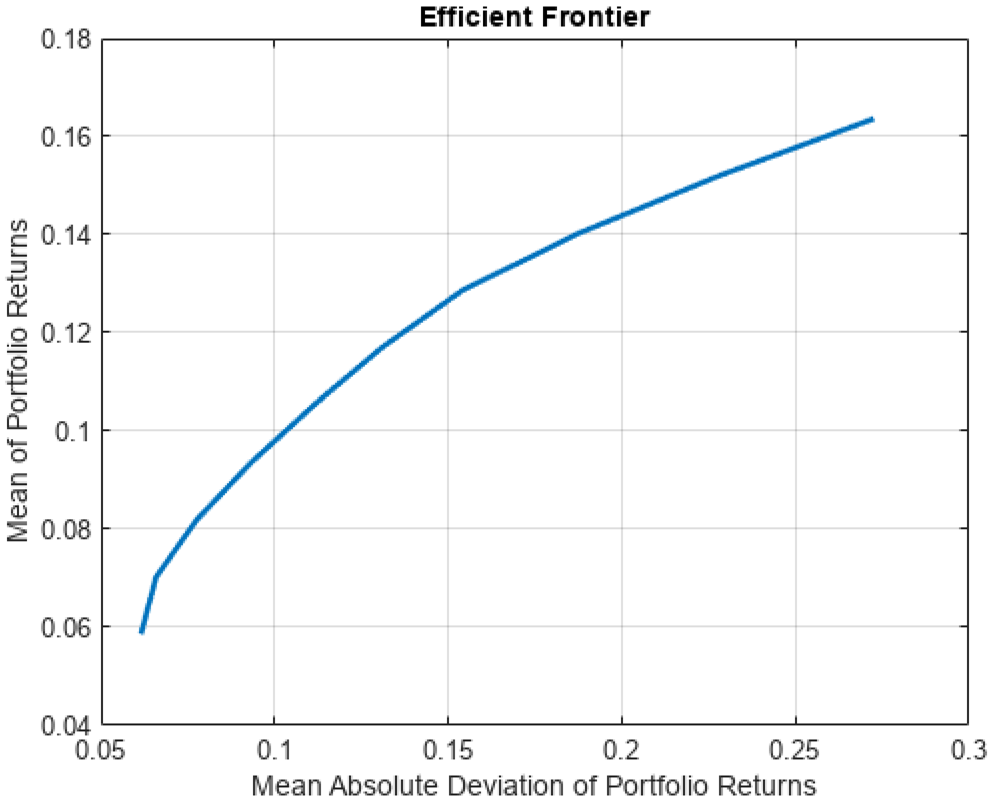
```
pwgt = 4×3

    0.1611         0         0
    0.4774    0.2137    0.0047
    0.1126    0.1384    0.1200
    0.2488    0.6480    0.8753
```

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 Using the Direct Method and Solver Options

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%. This example uses the default `'direct'` method to estimate the optimal portfolios with targeted portfolio risks. The `'direct'` method uses `fmincon` to solve the optimization problem that maximizes portfolio return, subject to the target risk as the quadratic nonlinear constraint. `setSolver` specifies the `solverType` and `SolverOptions` for `fmincon.`

```
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);

p = setSolver(p, 'fmincon', 'Display', 'off', 'Algorithm', 'sqp', ...
        'SpecifyObjectiveGradient', true, 'SpecifyConstraintGradient', true, ...
        'ConstraintTolerance', 1.0e-8, 'OptimalityTolerance', 1.0e-8, 'StepTolerance', 1.0e-8);
    plotFrontier(p);
```



```
pwgt = estimateFrontierByRisk(p, [0.12 0.20, 0.25]);

display(pwgt);

pwgt = 4×3

    0.1613    0.0000    0.0000
    0.4777    0.2139    0.0037
    0.1118    0.1381    0.1214
    0.2492    0.6480    0.8749
```

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`

Note If no initial portfolio is specified in `obj.InitPort`, it is assumed to be `0` so that `pbuy = max(0,pwgt)` and `psell = max(0,-pwgt)`. For more information on setting an initial portfolio, see `setInitPort`.

Data Types: `object`

`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`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[pwgt,pbuy,psell] = estimateFrontierByRisk(p,‘method’,‘direct’)`

`Method` — Method to estimate frontier by risk for `Portfolio` or `PortfolioMAD` objects
`'direct'` (default) | character vector with value `'direct'` or `'iterative'`

Method to estimate frontier by risk for `Portfolio` or `PortfolioMAD` objects, specified as the comma-separated pair consisting of `'Method'` and a character vector with one of the following values:

- `'direct'` — Construct one optimization problem to maximize the portfolio return with target risk as the nonlinear constraint, and solve it directly using `fmincon`, instead of iteratively exploring the efficient frontier. For an example of using the `'direct'` option, see “Obtain Portfolios with Targeted Portfolio Risks for a Portfolio Object Using the Direct Method and Solver Options” on page 15-890 and “Obtain Portfolios with Targeted Portfolio Risks for a PortfolioMAD Object Using the Direct Method and Solver Options” on page 15-892.
- `'iterative'` — One-dimensional optimization using `fminbnd` to find the portfolio return between min and max return on the frontier that minimizes the difference between the actual risk and target risk. Then the portfolio weights are obtained by solving a frontier by return problem. Consequently, the `'iterative'` method is slower than the `'direct'`.

Data Types: `char`

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);
```

or

```
[pwgt,pbuy,psell] = obj.estimateFrontierByRisk(TargetRisk,Name,Value);
```

Version History

Introduced in R2011a

See Also

estimateFrontier | estimateFrontierByReturn | estimateFrontierLimits | setInitPort | rng | setBounds | setMinMaxNumAssets

Topics

“Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-94
“Estimate Efficient Frontiers for Portfolio Object” on page 4-118
“Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-82
“Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-102
“Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object” on page 6-80

“Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-97
“Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
“Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215
“Portfolio Optimization Using Factor Models” on page 4-224
“Portfolio Optimization Theory” on page 4-3

estimateFrontierLimits

Estimate optimal portfolios at endpoints of efficient frontier

Syntax

```
[pwgt,pbuy,psell] = estimateFrontierLimits(obj)
[pwgt,pbuy,psell] = estimateFrontierLimits(obj,Choice)
```

Description

`[pwgt,pbuy,psell] = estimateFrontierLimits(obj)` estimates optimal portfolios at endpoints of efficient frontier for `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` objects. For details on the respective workflows when using these different objects, see “Portfolio Object Workflow” on page 4-17, “PortfolioCVaR Object Workflow” on page 5-16, and “PortfolioMAD Object Workflow” on page 6-15.

`[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 Portfolio Object with BoundType, MinNumAsset, and MaxNumAsset Constraints

When any one, or any combination of the constraints from 'Conditional' BoundType, MinNumAssets, and MaxNumAssets are active, the portfolio problem is formulated as mixed integer programming problem and the MINLP solver is used.

Create a Portfolio object for three assets.

```
AssetMean = [ 0.0101110; 0.0043532; 0.0137058 ];
AssetCovar = [ 0.00324625 0.00022983 0.00420395;
               0.00022983 0.00049937 0.00019247;
               0.00420395 0.00019247 0.00764097 ];
p = Portfolio('AssetMean', AssetMean, 'AssetCovar', AssetCovar);
p = setDefaultConstraints(p);
```

Use setBounds with semicontinuous constraints to set *xi* = 0 or 0.02 <= *xi* <= 0.5 for all *i* = 1,...NumAssets.

```
p = setBounds(p, 0.02, 0.7,'BoundType', 'Conditional', 'NumAssets', 3);
```

When working with a Portfolio object, the setMinMaxNumAssets function enables you to set up the limits on the number of assets invested (as known as cardinality) constraints. This sets the total number of allocated assets satisfying the Bound constraints that are between MinNumAssets and MaxNumAssets. By setting MinNumAssets = MaxNumAssets = 2, only two of the three assets are invested in the portfolio.

```
p = setMinMaxNumAssets(p, 2, 2);
```

Use estimateFrontierLimits to estimate the optimal portfolios at endpoints of the efficient frontier.

```
[pwgt, pbuy, psell] = estimateFrontierLimits(p,'Both')
```

```
pwgt = 3×2

    0.3000    0.3000
    0.7000         0
         0    0.7000


pbuy = 3×2

    0.3000    0.3000
    0.7000         0
         0    0.7000


psell = 3×2

     0     0
     0     0
     0     0
```

The estimateFrontierLimits function uses the MINLP solver to solve this problem. Use the setSolverMINLP function to configure the SolverType and options.

```
p.solverTypeMINLP

ans =
'OuterApproximation'

p.solverOptionsMINLP

ans = struct with fields:
                           MaxIterations: 1000
                    AbsoluteGapTolerance: 1.0000e-07
                    RelativeGapTolerance: 1.0000e-05
                  NonlinearScalingFactor: 1000
                  ObjectiveScalingFactor: 1000
                                 Display: 'off'
                           CutGeneration: 'basic'
                MaxIterationsInactiveCut: 30
                      ActiveCutTolerance: 1.0000e-07
                    IntMainSolverOptions: [1x1 optim.options.Intlinprog]
    NumIterationsEarlyIntegerConvergence: 30
```

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
    0.0606         0
    0.0456         0
    0.0484    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);
```

```
    0.8823         0
    0.0420         0
    0.0394         0
    0.0363    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.

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

Data Types: object

Choice — Indicator for which portfolios to obtain at extreme ends of efficient frontier

[] (default) | character vector with values 'Both', 'Min', 'Max' | string with values "Both", "Min", "Max"

(Optional) Indicator which portfolios to obtain at the extreme ends of the efficient frontier, specified as a character vector with values 'Both' or "Both", 'Min' or "Min", or 'Max' or "Max". The options for a Choice action are:

- [] — Compute both minimum-risk and maximum-return portfolios.
- 'Both' or "Both" — Compute both minimum-risk and maximum-return portfolios.
- 'Min' or "Min" — Compute minimum-risk portfolio only.
- 'Max' or "Max" — Compute maximum-return portfolio only.

Data Types: char | string

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);
```

Version History

Introduced in R2011a

See Also

`estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierByRisk` | `rng` | `setBounds` | `setMinMaxNumAssets`

Topics

“Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-94
“Estimate Efficient Frontiers for Portfolio Object” on page 4-118
“Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-82
“Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-102
“Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object” on page 6-80
“Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-97
“Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
“Portfolio Optimization Theory” on page 4-3

estimateMaxSharpeRatio

Estimate efficient portfolio to maximize Sharpe ratio for Portfolio object

Syntax

```
[pwgt,pbuy,psell] = estimateMaxSharpeRatio(obj)
[pwgt,pbuy,psell] = estimateMaxSharpeRatio( ___ ,Name,Value)
```

Description

`[pwgt,pbuy,psell] = estimateMaxSharpeRatio(obj)` estimates efficient portfolio to maximize Sharpe ratio for `Portfolio` object. For details on the workflow, see “Portfolio Object Workflow” on page 4-17.

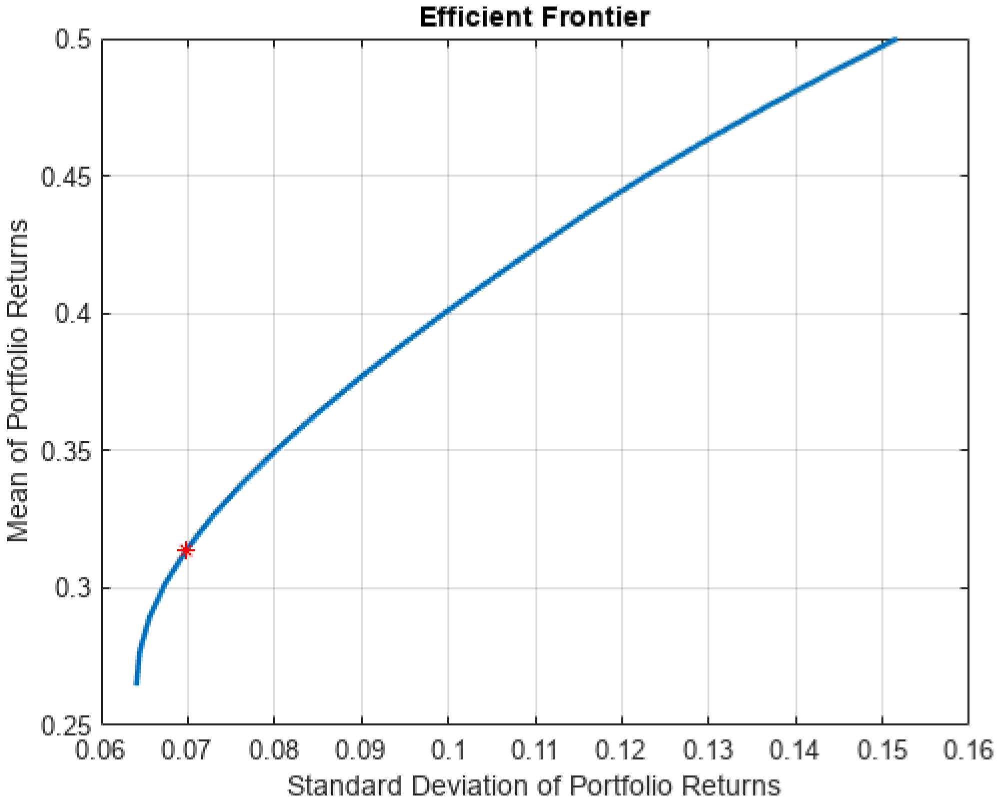
`[pwgt,pbuy,psell] = estimateMaxSharpeRatio( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Estimate Efficient Portfolio that Maximizes the Sharpe Ratio for a Portfolio Object

Estimate the efficient portfolio that maximizes the Sharpe ratio. The `estimateMaxSharpeRatio` function maximizes the Sharpe ratio among portfolios on the efficient frontier. This example uses the default `'direct'` method to estimate the maximum Sharpe ratio. For more information on the `'direct'` method, see “Algorithms” on page 15-912.

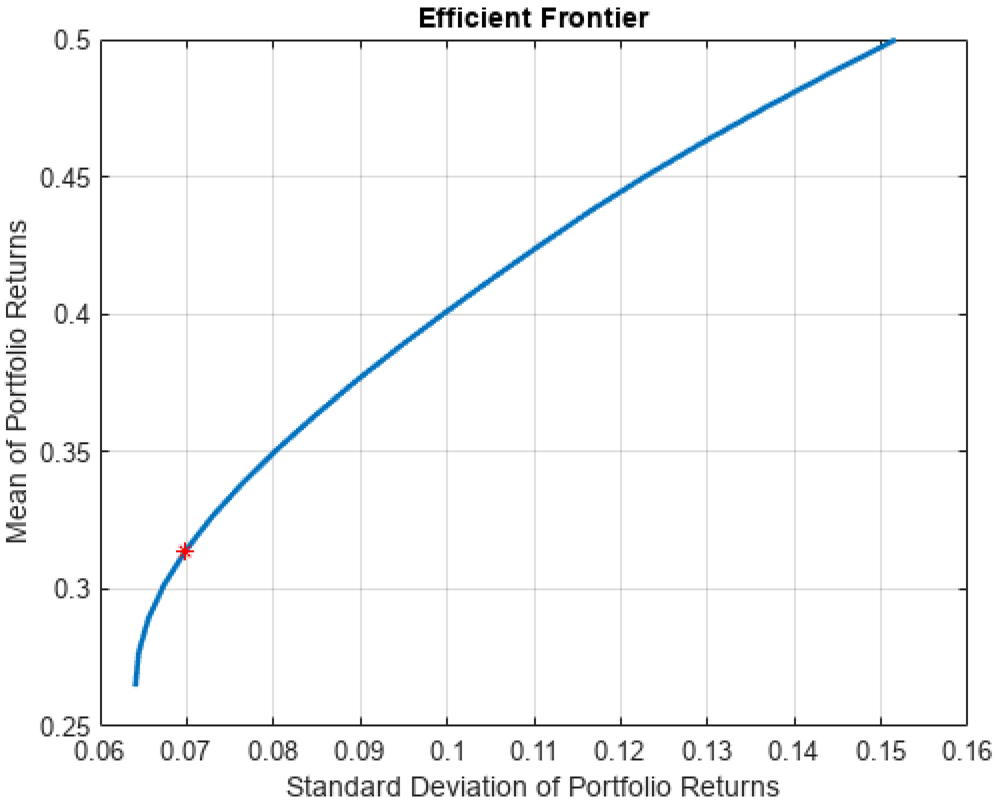
```
p = Portfolio('AssetMean',[0.3, 0.1, 0.5], 'AssetCovar',...
[0.01, -0.010,  0.004; -0.010,  0.040, -0.002;  0.004, -0.002,  0.023]);
p = setDefaultConstraints(p);
plotFrontier(p, 20);
weights = estimateMaxSharpeRatio(p);
[risk, ret] = estimatePortMoments(p, weights);
hold on
plot(risk,ret,'*r');
```



Estimate Efficient Portfolio that Maximizes the Sharpe Ratio for a Portfolio Object Using Solver Options with the Direct Method

Estimate the efficient portfolio that maximizes the Sharpe ratio. The `estimateMaxSharpeRatio` function maximizes the Sharpe ratio among portfolios on the efficient frontier. This example uses the `'direct'` method for a `Portfolio` object (`p`) that does not specify a tracking error and only uses linear constraints. The `setSolver` function is used to control the `SolverType` and `SolverOptions`. In this case, the `SolverType` is `quadprog`. For more information on the `'direct'` method, see “Algorithms” on page 15-912.

```
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);
p = setSolver(p,'quadprog','Display','off','ConstraintTolerance',1.0e-8,'OptimalityTolerance',1.0
weights = estimateMaxSharpeRatio(p);
[risk, ret] = estimatePortMoments(p, weights);
hold on
plot(risk,ret,'*r');
```

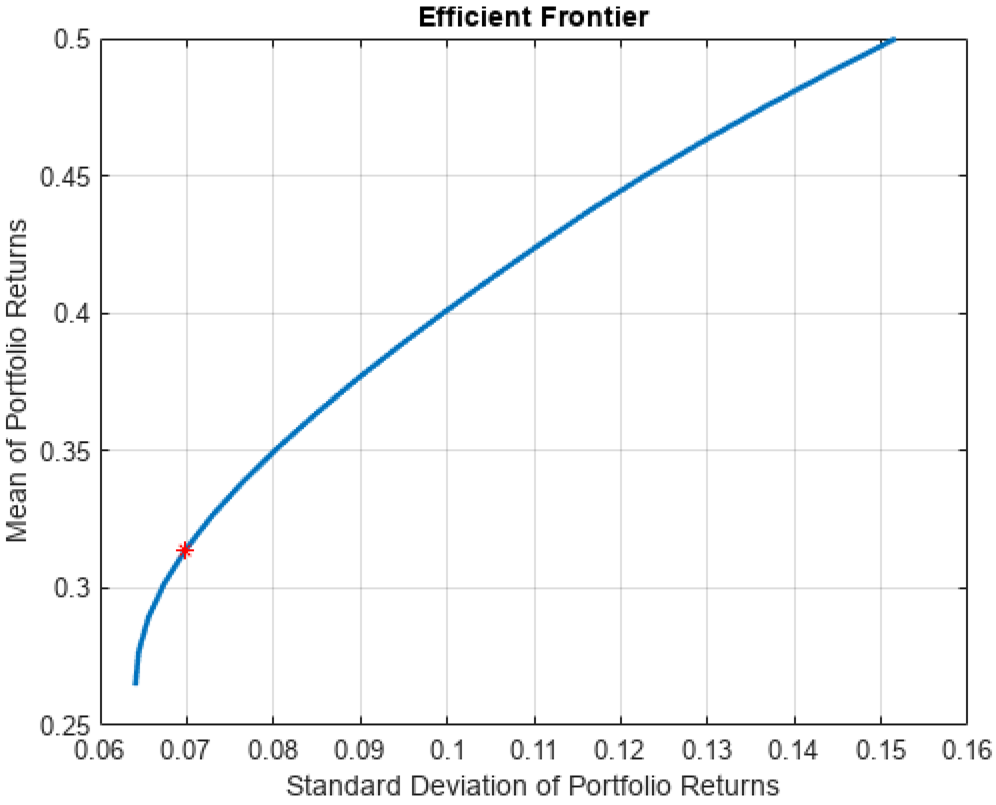


Estimate Efficient Portfolio that Maximizes the Sharpe Ratio for a Portfolio Object With Tracking Error Using the Direct Method and Solver Options

Estimate the efficient portfolio that maximizes the Sharpe ratio. The `estimateMaxSharpeRatio` function maximizes the Sharpe ratio among portfolios on the efficient frontier. This example uses the `'direct'` method for a `Portfolio` object (`p`) that specifies a tracking error uses nonlinear constraints. The `setSolver` function is used to control the `SolverType` and `SolverOptions`. In this case `fmincon` is the `SolverType`.

```
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],'lb', 0,'budget', 1);
plotFrontier(p, 20);

p = setSolver(p, 'fmincon', 'Display', 'off', 'Algorithm', 'sqp', ...
        'SpecifyObjectiveGradient', true, 'SpecifyConstraintGradient', true, ...
        'ConstraintTolerance', 1.0e-8, 'OptimalityTolerance', 1.0e-8, 'StepTolerance', 1.0e-8);

weights = estimateMaxSharpeRatio(p);

te = 0.08;
p = setTrackingError(p,te,weights);

[risk, ret] = estimatePortMoments(p,weights);
```

```
hold on
plot(risk,ret,'*r');
```



Estimate Efficient Portfolio that Maximizes the Sharpe Ratio for a Portfolio Object With a Risk-Free Asset Using the Direct and Iterative Method

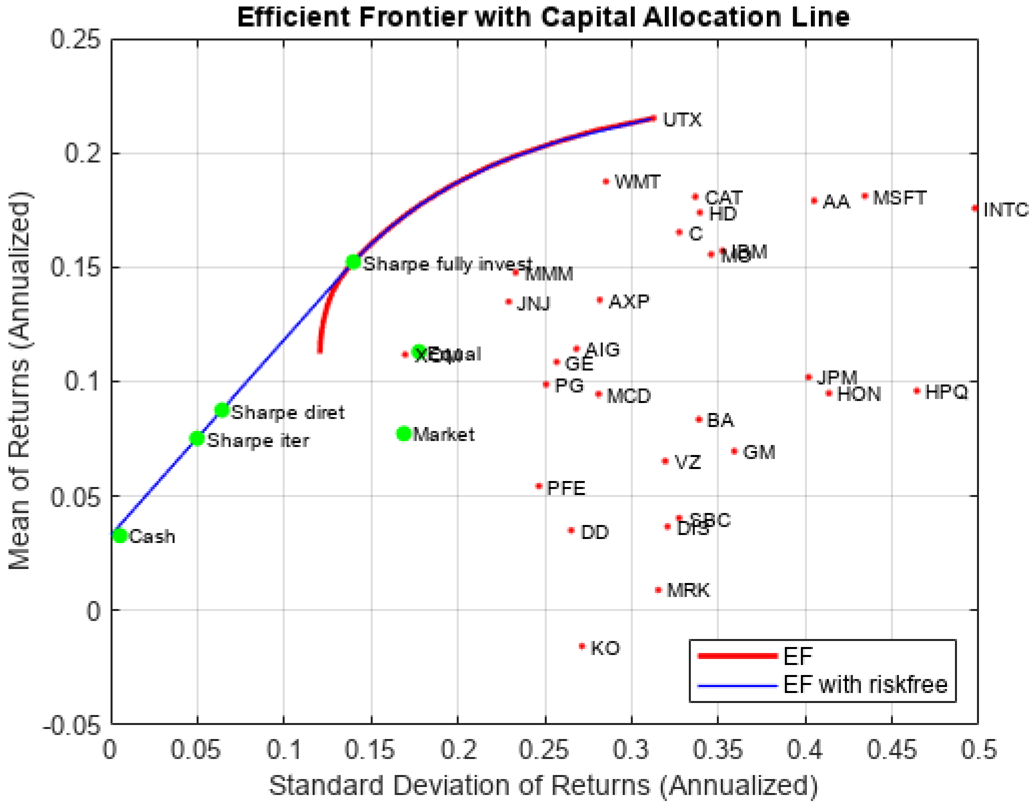
The `estimateMaxSharpeRatio` function maximizes the Sharpe ratio among portfolios on the efficient frontier. In the case of Portfolio with a risk-free asset, there are multiple efficient portfolios that maximize the Sharpe ratio on the capital asset line. Because of the nature of `'direct'` and `'iterative'` methods, the portfolio weights (`pwgts`) output from each of these methods might be different, but the Sharpe ratio is the same. This example demonstrates the scenario where the `pwgts` are different and the Sharpe ratio is the same.

```
load BlueChipStockMoments

mret = MarketMean;
mrsk = sqrt(MarketVar);
cret = CashMean;
crsk = sqrt(CashVar);

p = Portfolio('AssetList', AssetList, 'RiskFreeRate', CashMean);
p = setAssetMoments(p, AssetMean, AssetCovar);

p = setInitPort(p, 1/p.NumAssets);
```

```
[ersk, eret] = estimatePortMoments(p, p.InitPort);

p = setDefaultConstraints(p);
pwgt = estimateFrontier(p, 20);
[prsk, pret] = estimatePortMoments(p, pwgt);
pwgtshpr_fully = estimateMaxSharpeRatio(p,'Method','direct');
[riskshpr_fully, retshpr_fully] = estimatePortMoments(p,pwgtshpr_fully);

q = setBudget(p, 0, 1);
qwgt = estimateFrontier(q, 20);
[qrsk, qret] = estimatePortMoments(q, qwgt);
```

Plot the efficient frontier with a tangent line (0 to 1 cash).

```
pwgtshpr_direct = estimateMaxSharpeRatio(q,'Method','direct');
pwgtshpr_iter = estimateMaxSharpeRatio(q,'Method','iterative'); % Default for 'TolX' is 1e-8
[riskshpr_diret, retshpr_diret] = estimatePortMoments(q,pwgtshpr_direct);
[riskshpr_iter, retshpr_iter] = estimatePortMoments(q,pwgtshpr_iter);

clf;
portfolioexamples_plot('Efficient Frontier with Capital Allocation Line', ...
                {'line', prsk, pret, {'EF'}, '-r', 2}, ...
                {'line', qrsk, qret, {'EF with riskfree'}, '-b', 1}, ...
                {'scatter', [mrsk, crsk, ersk, riskshpr_fully, riskshpr_diret, riskshpr_iter], .
    [mret, cret, eret, retshpr_fully , retshpr_diret, retshpr_iter], {'Market', 'Cash', 'Equal',
                {'scatter', sqrt(diag(p.AssetCovar)), p.AssetMean, p.AssetList, '.r'});
```



When a risk-free asset is not available to the portfolio, or in other words, the portfolio is fully invested, the efficient frontier is curved, corresponding to the red line in the above figure. Therefore, there is a unique (risk, return) point that maximizes the Sharpe ratio, which the `'iterative'` and `'direct'` methods will both find. If the portfolio is allowed to invest in a risk-free asset, part of the red efficient frontier line is replaced by the capital allocation line, resulting in the efficient frontier of a portfolio with a risk-free investment (blue line). All the (risk, return) points on the straight blue line share the same Sharpe ratio. Also, it is likely that the `'iterative'` and `'direct'` methods end up with different points, therefore there are different portfolio allocations.

When using the `'iterative'` method, you can use an optional `'TolX'` name-value argument. `TolX` is a termination tolerance that is related to the possible return levels of the efficient frontier. If the selected `TolX` value is large compared to the range of returns, then the accuracy of the solution is poor. `TolX` should be a number smaller than 0.01*(`maxReturn` - `minReturn`).

```
maxReturn = max(qret); % Max return portfolio
minReturn = min(qret); % Min return portfolio
display(0.01*(maxReturn-minReturn))
```

```
   1.5193e-04
```

The purpose of increasing the termination tolerance is to speed up the convergence of the `'iterative'` algorithm. However, as previously mentioned, the accuracy of the solution will decrease. You can see this in the table that follows.

```
pwgtshpr_iter_largerTol = estimateMaxSharpeRatio(q, 'Method', 'iterative',...
    'TolX', 1e-4);
display(table(pwgtshpr_iter, pwgtshpr_iter_largerTol,...
    'VariableNames', {'Default TolX = 1e-8','TolX = 1e-4'}))
```

```
  30x2 table

    Default TolX = 1e-8    TolX = 1e-4
    ___________________    ___________

                  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
          0.0094789        0.0093895
           0.032042         0.031739
                  0                0
                  0                0
                  0                0
           0.055162         0.054642
           0.049805         0.049335
                  0                0
           0.015934         0.015784
           0                  0
    0.026542           0.026292
           0                  0
    0.021371           0.021169
           0                  0
    0.078468           0.077728
    0.067045           0.066412
```

In the table, the value of the weights varies slightly with respect to the weights obtained with the default tolerance, as expected.

Estimate Efficient Portfolio that Maximizes the Sharpe Ratio for a `Portfolio` Object with Semicontinuous and Cardinality Constraints

Create a `Portfolio` object for three assets.

```
AssetMean = [ 0.0101110; 0.0043532; 0.0137058 ];
AssetCovar = [ 0.00324625 0.00022983 0.00420395;
               0.00022983 0.00049937 0.00019247;
               0.00420395 0.00019247 0.00764097 ];
p = Portfolio('AssetMean', AssetMean, 'AssetCovar', AssetCovar);
p = setDefaultConstraints(p);
```

Use `setBounds` with semicontinuous constraints to set *xi* = `0` or `0.02 <= xi <= 0.5` for all *i* = `1`,...`NumAssets`.

```
p = setBounds(p, 0.02, 0.5,'BoundType', 'Conditional', 'NumAssets', 3);
```

When working with a `Portfolio` object, the `setMinMaxNumAssets` function enables you to set up cardinality constraints for a long-only portfolio. This sets the cardinality constraints for the `Portfolio` object, where the total number of allocated assets satisfying the nonzero semi-continuous constraints are between `MinNumAssets` and `MaxNumAssets`. By setting `MinNumAssets` = `MaxNumAssets` = 2, only two of the three assets are invested in the portfolio.

```
p = setMinMaxNumAssets(p, 2, 2);
```

Use `estimateMaxSharpeRatio` to estimate efficient portfolio to maximize Sharpe ratio.

```
weights = estimateMaxSharpeRatio(p,'Method','iterative')
```

```
weights = 3×1

         0
    0.5000
    0.5000
```

The `estimateMaxSharpeRatio` function uses the MINLP solver to solve this problem. Use the `setSolverMINLP` function to configure the `SolverType` and options.

```
p.solverOptionsMINLP
```

```
ans = struct with fields:
                           MaxIterations: 1000
                    AbsoluteGapTolerance: 1.0000e-07
```

```
               RelativeGapTolerance: 1.0000e-05
              NonlinearScalingFactor: 1000
              ObjectiveScalingFactor: 1000
                             Display: 'off'
                       CutGeneration: 'basic'
             MaxIterationsInactiveCut: 30
                 ActiveCutTolerance: 1.0000e-07
               IntMainSolverOptions: [1x1 optim.options.Intlinprog]
   NumIterationsEarlyIntegerConvergence: 30
```

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`. If the max return of portfolio is less than the `RiskFreeRate`, the solution is set as `pwgt` at max return and the resulting Sharpe ratio will be negative.

For more information on creating a portfolio object, see

- `Portfolio`

Data Types: `object`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[pwgt,pbuy,psell] = estimateMaxSharpeRatio(p,'Method’,'iterative')`

Method — Method to estimate Sharpe ratio

`'direct'` (default) | character vector with value `'direct'` or `'iterative'`

Method to estimate Sharpe ratio, specified as the comma-separated pair consisting of `'Method'` and a character vector with one of the following values:

- `'direct'` — Transform the Sharpe ratio function into a quadratic one and solve one optimization problem directly, instead of iteratively exploring the efficient frontier. The `'direct'` option either uses the solver `quadprog` (for a problem with linear constraints) or `fmincon` (for a problem with nonlinear constraints). For an example of using the `'direct'` option, see “Estimate Efficient Portfolio that Maximizes the Sharpe Ratio for a Portfolio Object Using Solver Options with the Direct Method” on page 15-904 and “Estimate Efficient Portfolio that Maximizes the Sharpe Ratio for a Portfolio Object With Tracking Error Using the Direct Method and Solver Options” on page 15-905.

- 'iterative' — One-dimensional optimization using fminbnd to find the portfolio that maximizes the Sharpe ratio by iteratively exploring the efficient frontier.

Note If you are using estimateMaxSharpeRatio with a Portfolio object with semicontinuous and cardinality constraints specified by setBounds and setMinMaxNumAssets, you can only use the 'iterative' method.

Data Types: char

TolX — Tolerance on the step size for fminbnd convergence when using 'iterative' method
1e-8 (default) | positive scalar

Tolerance on the step size for fminbnd convergence when using 'iterative' method, specified as the comma-separated pair consisting of 'TolX' and a positive scalar. The step size is related to the return levels achieved on the efficient frontier and tried by fminbnd. If the specified TolX is large compared to the range of returns, the accuracy of the solution is poor. TolX should be a number smaller than 0.01*(maxReturn - minReturn).

Note If the 'direct' method is selected, TolX is ignored.

Data Types: double

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).

More About

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 `estimateMaxSharpeRatio` 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 either using the `'direct'` or `'iterative'` method. For the `'direct'` method, consider the following scenario. To maximize the Sharpe ratio is to:

$$Maximize\frac{\mu^T x - r_f}{\sqrt{x^T C x}}, s.t. \sum x_i = 1, \;\; 0 \le x_i \le 1,$$

where μ and C are the mean and covariance matrix, and r_f is the risk-free rate.

If μ^T x - $r_f \le 0$ for all x the portfolio that maximizes the Sharpe ratio is the one with maximum return.

If μ^T x - $r_f > 0$, let $t = \dfrac{1}{\mu^T x - r_f}$

and $y = tx$ (Cornuejols [1] section 8.2). Then after some substitutions, you can transform the original problem into the following form,

$$Minimize\ y^T C y,\ s.t.\ \sum y_i = t,\ t > 0,\ 0 \le y_i \le \mathrm{t}\ ,\ \mu^T y - r_f \mathrm{t} = 1.$$

Only one optimization needs to be solved, hence the name “direct”. The portfolio weights can be recovered by $x^* = y^* / t^*$.

For the `'iterative'` method, the idea is to iteratively explore the portfolios at different return levels on the efficient frontier and locate the one with maximum Sharpe ratio. Therefore, multiple optimization problems are solved during the process, instead of only one in the `'direct'` method. Consequently, the `'iterative'` method is slow compared to `'direct'` method.

Version History

Introduced in R2011b

References

[1] Cornuejols, G. and Reha Tütüncü. *Optimization Methods in Finance*. Cambridge University Press, 2007.

See Also

`estimatePortSharpeRatio` | `estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierByRisk` | `setBounds` | `setMinMaxNumAssets`

Topics

“Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-94
“Working with 'Conditional' BoundType, MinNumAssets, and MaxNumAssets Constraints Using Portfolio Objects” on page 4-78
“Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
“Bond Portfolio Optimization Using Portfolio Object” on page 10-30
“Portfolio Optimization Theory” on page 4-3

estimatePortSharpeRatio

Estimate Sharpe ratio of given portfolio weights for Portfolio object

Syntax

```
psharpe = estimatePortSharpeRatio(obj,pwgt)
```

Description

`psharpe = estimatePortSharpeRatio(obj,pwgt)` estimates the Sharpe ratio of given portfolio weights for a `Portfolio` object. For details on the workflow, see “Portfolio Object Workflow” on page 4-17.

Examples

Estimate Sharpe Ratios of the Given Portfolio Weights

This example shows how to find efficient portfolios that satisfy the target returns and then find the Sharpe ratios corresponding to each of the portfolios.

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%. Use `estimatePortSharpeRatio` to obtain the Sharpe ratio for the collection of portfolios (`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 = estimateFrontierByReturn(p, [0.06, 0.09, 0.12]);

display(pwgt);
```

```
pwgt = 4×3

    0.8772    0.5032    0.1293
    0.0434    0.2488    0.4541
    0.0416    0.0780    0.1143
    0.0378    0.1700    0.3022
```

`pwgt` is a `NumAssets`-by-`NumPorts` matrix where `NumAssets` is the number of asset in the universe and `NumPorts` is the number of portfolios in the collection of portfolios.

```
psharpe = estimatePortSharpeRatio(p,pwgt)
psharpe = 3×1

    0.7796
    0.8519
    0.7406
```

psharpe is a NumPorts-by-1 vector, where NumPorts is the number of portfolios in the collection of portfolios.

Estimate Sharpe Ratios of the Given Portfolio Weights Portfolio Object with Integrality Constraints

Create a Portfolio object for three assets.

```
AssetMean = [ 0.0101110; 0.0043532; 0.0137058 ];
AssetCovar = [ 0.00324625 0.00022983 0.00420395;
               0.00022983 0.00049937 0.00019247;
               0.00420395 0.00019247 0.00764097 ];
p = Portfolio('AssetMean', AssetMean, 'AssetCovar', AssetCovar);
p = setDefaultConstraints(p);
```

Use setBounds with semi-continuous constraints to set *xi*=0 or 0.02<=*xi*<=0.5 for all *i*=1,...NumAssets.

```
p = setBounds(p, 0.02, 0.5,'BoundType', 'Conditional', 'NumAssets', 3);
```

When working with a Portfolio object, the setMinMaxNumAssets function enables you to set up cardinality constraints for a long-only portfolio. This sets the cardinality constraints for the Portfolio object, where the total number of allocated assets satisfying the nonzero semi-continuous constraints are between MinNumAssets and MaxNumAssets. By setting MinNumAssets=MaxNumAssets=2, only two of the three assets are invested in the portfolio.

```
p = setMinMaxNumAssets(p, 2, 2);
```

Use estimatePortSharpeRatio to estimate Sharpe ratio of given portfolio weights for a Portfolio object.

```
pwgt = estimateFrontierByReturn(p, [ 0.0072321, 0.0119084 ]);
psharpe = estimatePortSharpeRatio(p,pwgt)
```

```
psharpe = 2×1

    0.2230
    0.1715
```

The estimatePortSharpeRatio function uses the MINLP solver to solve this problem. Use the setSolverMINLP function to configure the SolverType and options.

```
p.solverOptionsMINLP
```

```
ans = struct with fields:
                           MaxIterations: 1000
                    AbsoluteGapTolerance: 1.0000e-07
```

```
                     RelativeGapTolerance: 1.0000e-05
                   NonlinearScalingFactor: 1000
                   ObjectiveScalingFactor: 1000
                                  Display: 'off'
                            CutGeneration: 'basic'
                 MaxIterationsInactiveCut: 30
                       ActiveCutTolerance: 1.0000e-07
                     IntMainSolverOptions: [1x1 optim.options.Intlinprog]
     NumIterationsEarlyIntegerConvergence: 30
```

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.

Data Types: object

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

psharpe — Sharpe ratios of the given portfolios
vector

Sharpe ratios of the given portfolios, returned as a NumPorts-by-1 vector.

More About

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 for each portfolio in pwgt.

estimatePortSharpeRatio computes the Sharpe ratio with mean and standard deviation (which is the square-root of variance) of portfolio returns.

Tips

You can also use dot notation to estimate the Sharpe ratio of given portfolio weights.

```
psharpe = obj.estimatePortSharpeRatio(pwgt);
```

Version History

Introduced in R2018a

See Also

`estimateMaxSharpeRatio` | `estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierByRisk`

Topics

“Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-94
“Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
“Portfolio Optimization Theory” on page 4-3

estimatePortMoments

Estimate moments of portfolio returns for Portfolio object

Syntax

```
[prsk,pret] = estimatePortMoments(obj,pwgt)
```

Description

`[prsk,pret] = estimatePortMoments(obj,pwgt)` estimate moments of portfolio returns for a `Portfolio` object. For details on the workflow, see "Portfolio Object Workflow" on page 4-17.

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
```

Identify the Range of Risks and Returns for Efficient Portfolios `Portfolio` Object with Integrality Constraints

Create a `Portfolio` object for three assets.

```
AssetMean = [ 0.0101110; 0.0043532; 0.0137058 ];
AssetCovar = [ 0.00324625 0.00022983 0.00420395;
               0.00022983 0.00049937 0.00019247;
               0.00420395 0.00019247 0.00764097 ];
```

```
p = Portfolio('AssetMean', AssetMean, 'AssetCovar', AssetCovar);
p = setDefaultConstraints(p);
```

Use `setBounds` with semi-continuous constraints to set *xi*=`0` or `0.02<=`*xi*`<=0.5` for all *i*=1,...`NumAssets`.

```
p = setBounds(p, 0.02, 0.5,'BoundType', 'Conditional', 'NumAssets', 3);
```

When working with a `Portfolio` object, the `setMinMaxNumAssets` function enables you to set up cardinality constraints for a long-only portfolio. This sets the cardinality constraints for the `Portfolio` object, where the total number of allocated assets satisfying the nonzero semi-continuous constraints are between `MinNumAssets` and `MaxNumAssets`. By setting `MinNumAssets=MaxNumAssets=`2, only two of the three assets are invested in the portfolio.

```
p = setMinMaxNumAssets(p, 2, 2);
```

Use `estimatePortMoments` to estimate moments of portfolio returns for a `Portfolio` object.

```
pwgt = estimateFrontierLimits(p);
[prsk, pret] = estimatePortMoments(p, pwgt)
```

```
prsk = 2×1

    0.0324
    0.0695


pret = 2×1

    0.0072
    0.0119
```

The `estimatePortMoments` function uses the MINLP solver to solve this problem. Use the `setSolverMINLP` function to configure the `SolverType` and options.

```
p.solverOptionsMINLP
```

```
ans = struct with fields:
                           MaxIterations: 1000
                    AbsoluteGapTolerance: 1.0000e-07
                    RelativeGapTolerance: 1.0000e-05
                  NonlinearScalingFactor: 1000
                  ObjectiveScalingFactor: 1000
                                 Display: 'off'
                           CutGeneration: 'basic'
                MaxIterationsInactiveCut: 30
                      ActiveCutTolerance: 1.0000e-07
                    IntMainSolverOptions: [1x1 optim.options.Intlinprog]
    NumIterationsEarlyIntegerConvergence: 30
```

Calculate Portfolio Weights For a Target Risk Level

The `Portfolio` object is able to find an efficient portfolio with respect to a specified target risk.

Load Portfolio

Load a vector of expected returns and a covariance matrix for 30 stocks.

```
load StockStats
```

Create `Portfolio` Object with Default Constraints

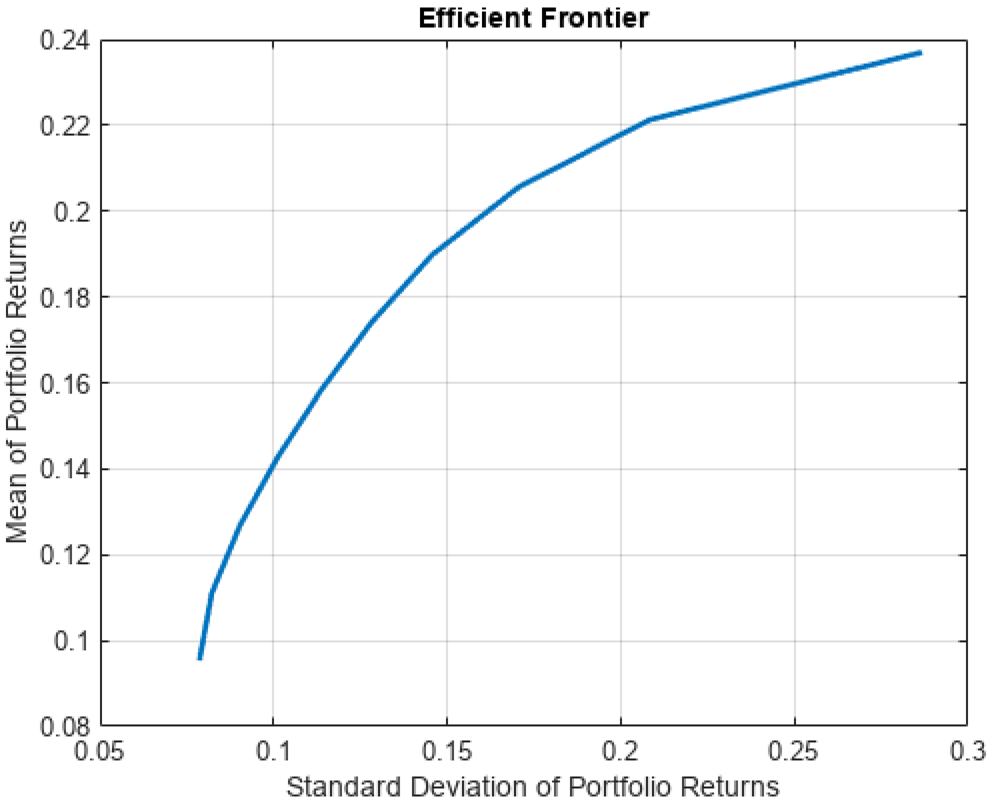
The default constraints for a `Portfolio` object are that it is a long-only portfolio and that it is 100% invested. Many other constraint types are possible.

```
P = Portfolio('mean',expRet,'covar',expCov);
P = setDefaultConstraints(P);
```

Plot Full Efficient Frontier

Use the `plotFrontier` function with the `Portfolio` object to visualize the full frontier. Other functions exist that allow you to probe into particular portfolios along the frontier.

```
P.plotFrontier
```



Capture Upper and Lower Bounds of Portfolio Risks and Returns

It is useful to know what are the upper and lower limits of the portfolio moments along the efficient frontier. This information allows you to determine what are feasible targets. Use the `estimateFrontierLimits` function with the `Portfolio` object to identify the weights at these extremes. Then you can use the `estimatePortMoments` function `Portfolio` object to find the limiting moments.

```
P_Weights1 = P.estimateFrontierLimits;
[P_risklimits, P_returnlimits] = P.estimatePortMoments(P_Weights1)
```

```
P_risklimits = 2×1

    0.0786
    0.2868


P_returnlimits = 2×1

    0.0954
    0.2370
```

Estimate Efficient Portfolio with Target Return

Select a target return for the portfolio somewhere in the feasible region. You can estimate its makeup with `estimateFrontierByRisk` and then confirm its moments using `estimatePortMoments`.

```
targetReturn = 0.15;

P_Weights2 = P.estimateFrontierByReturn(targetReturn);
[P_risk2, P_return2] = P.estimatePortMoments(P_Weights2)
```

```
P_risk2 = 0.1068

P_return2 = 0.1500
```

The return matches the `targetReturn` value and the risk is in agreement with the efficient frontier plot.

Estimate Efficient Portfolio with Target Risk

The `Portfolio` object is able to find an efficient portfolio with a specified target risk.

```
targetRisk = 0.2;

P_Weights3 = P.estimateFrontierByRisk(targetRisk);
[P_risk3, P_return3] = P.estimatePortMoments(P_Weights3)
```

```
P_risk3 = 0.2000

P_return3 = 0.2182
```

Another useful feature of the `estimateFrontierByReturn` and `estimateFrontierByRisk` functions is what happens if you specify an infeasible (too high or too low) target. In this case, these functions provide a warning message to suggest the best solution.

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`.

Data Types: `object`

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);
```

Version History

Introduced in R2011a

See Also

estimatePortReturn | estimatePortRisk

Topics

“Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-94
“Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
“Portfolio Optimization Theory” on page 4-3

estimatePortReturn

Estimate mean of portfolio returns

Syntax

```
pret = estimatePortReturn(obj,pwgt)
```

Description

`pret = estimatePortReturn(obj,pwgt)` estimates the mean of portfolio returns (as the proxy for portfolio return) for `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` objects. For details on the respective workflows when using these different objects, see “Portfolio Object Workflow” on page 4-17, “PortfolioCVaR Object Workflow” on page 5-16, and “PortfolioMAD Object Workflow” on page 6-15.

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 `Portfolio` Object with Integrality Constraints

Create a `Portfolio` object for three assets.

```
AssetMean = [ 0.0101110; 0.0043532; 0.0137058 ];
AssetCovar = [ 0.00324625 0.00022983 0.00420395;
               0.00022983 0.00049937 0.00019247;
               0.00420395 0.00019247 0.00764097 ];
p = Portfolio('AssetMean', AssetMean, 'AssetCovar', AssetCovar);
p = setDefaultConstraints(p);
```

Use setBounds with semi-continuous constraints to set *xi*=0 or 0.02<=*xi*<=0.5 for all *i*=1,...NumAssets.

```
p = setBounds(p, 0.02, 0.5,'BoundType', 'Conditional', 'NumAssets', 3);
```

When working with a Portfolio object, the setMinMaxNumAssets function enables you to set up cardinality constraints for a long-only portfolio. This sets the cardinality constraints for the Portfolio object, where the total number of allocated assets satisfying the nonzero semi-continuous constraints are between MinNumAssets and MaxNumAssets. By setting MinNumAssets=MaxNumAssets=2, only two of the three assets are invested in the portfolio.

```
p = setMinMaxNumAssets(p, 2, 2);
```

Use estimatePortReturn to estimate the mean of portfolio returns for a Portfolio object.

```
pwgt = estimateFrontierLimits(p);
pret = estimatePortReturn(p, pwgt)
```

```
pret = 2×1

    0.0072
    0.0119
```

The estimatePortReturn function uses the MINLP solver to solve this problem. Use the setSolverMINLP function to configure the SolverType and options.

```
p.solverOptionsMINLP
```

```
ans = struct with fields:
                           MaxIterations: 1000
                    AbsoluteGapTolerance: 1.0000e-07
                    RelativeGapTolerance: 1.0000e-05
                  NonlinearScalingFactor: 1000
                  ObjectiveScalingFactor: 1000
                                 Display: 'off'
                           CutGeneration: 'basic'
                MaxIterationsInactiveCut: 30
                      ActiveCutTolerance: 1.0000e-07
                    IntMainSolverOptions: [1x1 optim.options.Intlinprog]
    NumIterationsEarlyIntegerConvergence: 30
```

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.

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

Data Types: object

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).

Note Depending on whether costs have been set, the portfolio return is either gross or net portfolio returns. For information on setting costs, see setCosts.

Tips

You can also use dot notation to estimate the mean of portfolio returns (as the proxy for portfolio return).

```
pret = obj.estimatePortReturn(pwgt);
```

Version History

Introduced in R2011a

See Also

estimatePortRisk | estimateFrontierByReturn | estimateFrontierByRisk | rng

Topics

“Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-94
“Estimate Efficient Frontiers for Portfolio Object” on page 4-118
“Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-82
“Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-102
“Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object” on page 6-80
“Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-97

“Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
“Portfolio Optimization Theory” on page 4-3

estimatePortRisk

Estimate portfolio risk according to risk proxy associated with corresponding object

Syntax

```
prsk = estimatePortRisk(obj,pwgt)
```

Description

`prsk = estimatePortRisk(obj,pwgt)` estimates portfolio risk according to the risk proxy associated with the corresponding object (`obj`) for `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` objects. For details on the respective workflows when using these different objects, see "Portfolio Object Workflow" on page 4-17, "PortfolioCVaR Object Workflow" on page 5-16, and "PortfolioMAD Object Workflow" on page 6-15.

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
```

Standard Deviation of Portfolio Returns as the Proxy for Portfolio Risk for a `Portfolio` Object with Integrality Constraints

Create a `Portfolio` object for three assets.

```
AssetMean = [ 0.0101110; 0.0043532; 0.0137058 ];
AssetCovar = [ 0.00324625 0.00022983 0.00420395;
               0.00022983 0.00049937 0.00019247;
               0.00420395 0.00019247 0.00764097 ];
```

```
p = Portfolio('AssetMean', AssetMean, 'AssetCovar', AssetCovar);
p = setDefaultConstraints(p);
```

Use `setBounds` with semi-continuous constraints to set *xi*=0 or 0.02<=*xi*<=0.5 for all *i*=1,...NumAssets.

```
p = setBounds(p, 0.02, 0.5,'BoundType', 'Conditional', 'NumAssets', 3);
```

When working with a `Portfolio` object, the `setMinMaxNumAssets` function enables you to set up cardinality constraints for a long-only portfolio. This sets the cardinality constraints for the `Portfolio` object, where the total number of allocated assets satisfying the nonzero semi-continuous constraints are between `MinNumAssets` and `MaxNumAssets`. By setting `MinNumAssets`=`MaxNumAssets`=2, only two of the three assets are invested in the portfolio.

```
p = setMinMaxNumAssets(p, 2, 2);
```

Use `estimatePortRisk` to estimate the portfolio risk according to a risk proxy associated with a `Portfolio` object.

```
pwgt = estimateFrontierLimits(p);
prsk = estimatePortRisk(p, pwgt)
```

```
prsk = 2×1

    0.0324
    0.0695
```

The `estimatePortRisk` function uses the MINLP solver to solve this problem. Use the `setSolverMINLP` function to configure the `SolverType` and options.

```
p.solverOptionsMINLP
```

```
ans = struct with fields:
                           MaxIterations: 1000
                    AbsoluteGapTolerance: 1.0000e-07
                    RelativeGapTolerance: 1.0000e-05
                  NonlinearScalingFactor: 1000
                  ObjectiveScalingFactor: 1000
                                 Display: 'off'
                           CutGeneration: 'basic'
                MaxIterationsInactiveCut: 30
                      ActiveCutTolerance: 1.0000e-07
                    IntMainSolverOptions: [1x1 optim.options.Intlinprog]
    NumIterationsEarlyIntegerConvergence: 30
```

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.

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

Data Types: object

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);
```

Version History

Introduced in R2011a

See Also

estimatePortStd | estimatePortVaR | estimateFrontierByReturn | estimateFrontierByRisk | rng

Topics

"Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object" on page 4-94

“Estimate Efficient Frontiers for Portfolio Object” on page 4-118
“Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object” on page 5-82
“Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-102
“Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object” on page 6-80
“Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-97
“Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
“Portfolio Optimization Theory” on page 4-3

estimatePortStd

Estimate standard deviation of portfolio returns

Syntax

```
pstd = estimatePortStd(obj,pwgt)
```

Description

`pstd = estimatePortStd(obj,pwgt)` estimate standard deviation of portfolio returns for `PortfolioCVaR` or `PortfolioMAD` objects. For details on the workflows, see “PortfolioCVaR Object Workflow” on page 5-16 and “PortfolioMAD Object Workflow” on page 6-15.

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.

Input Arguments

obj — Object for portfolio
object

Object for portfolio, specified using a PortfolioCVaR or PortfolioMADobject.

For more information on creating a PortfolioCVaR or PortfolioMAD object, see

- PortfolioCVaR
- PortfolioMAD

Data Types: object

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);
```

Version History

Introduced in R2012b

See Also

`estimatePortReturn` | `estimateFrontierByReturn` | `estimateFrontierByRisk` | `estimatePortVaR` | `rng`

Topics

"Estimate Efficient Frontiers for PortfolioCVaR Object" on page 5-102
"Estimate Efficient Frontiers for PortfolioMAD Object" on page 6-97
"Portfolio Optimization Theory" on page 4-3

External Websites

Getting Started with Portfolio Optimization (4 min 13 sec)
CVaR Portfolio Optimization (4 min 56 sec)

estimatePortVaR

Estimate value-at-risk for PortfolioCVaR object

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`. For details on the workflow, see “PortfolioCVaR Object Workflow” on page 5-16.

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.

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`

Data Types: `object`

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);
```

Version History

Introduced in R2012b

See Also

`estimatePortStd` | `setProbabilityLevel` | `rng`

Topics

“Estimate Efficient Frontiers for PortfolioCVaR Object” on page 5-102
“Conditional Value-at-Risk” on page 5-5

External Websites

Getting Started with Portfolio Optimization (4 min 13 sec)
CVaR Portfolio Optimization (4 min 56 sec)

estimateScenarioMoments

Estimate mean and covariance of asset return scenarios

Syntax

```
[ScenarioMean,ScenarioCovar] = estimateScenarioMoments(obj)
```

Description

`[ScenarioMean,ScenarioCovar] = estimateScenarioMoments(obj)` estimates mean and covariance of asset return scenarios for `PortfolioCVaR` or `PortfolioMAD` objects. For details on the workflows, see “PortfolioCVaR Object Workflow” on page 5-16, and “PortfolioMAD Object Workflow” on page 6-15.

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 = 4×1

    0.0039
    0.0082
    0.0102
    0.0154


ScenarioCovar = 4×4

    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 = 4×1

    0.0039
    0.0082
    0.0102
    0.0154


ScenarioCovar = 4×4

    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.

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

Data Types: object

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
```

Version History

Introduced in R2012b

See Also

setScenarios | estimatePortRisk | simulateNormalScenariosByMoments | rng

Topics

"Asset Returns and Scenarios Using PortfolioCVaR Object" on page 5-36
"Asset Returns and Scenarios Using PortfolioMAD Object" on page 6-34

External Websites

Getting Started with Portfolio Optimization (4 min 13 sec)
CVaR Portfolio Optimization (4 min 56 sec)

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 = 1×2

    0.1995    0.1002


ExpCovariance = 2×2

    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 $\mathtt{DecayFactor}^k$. 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 determining the expected return and covariance from a return time series.

Version History

Introduced before R2006a

See Also

`cov` | `mean` | `cov2corr`

Topics

"Portfolio Optimization Functions" on page 3-3

fanplot

Plot combined historical and forecast data to visualize possible outcomes

Syntax

```
fanplot(historical,forecast)
fanplot( ___ ,Name,Value)
```

```
fanplot(ax,historical,forecast)
fanplot( ___ ,Name,Value)
```

```
h = fanplot(ax,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.

`fanplot(ax,historical,forecast)` generates a fan chart using an optional `ax` argument.

`fanplot( ___ ,Name,Value)` generates a fan chart using optional name-value pair arguments.

`h = fanplot(ax,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

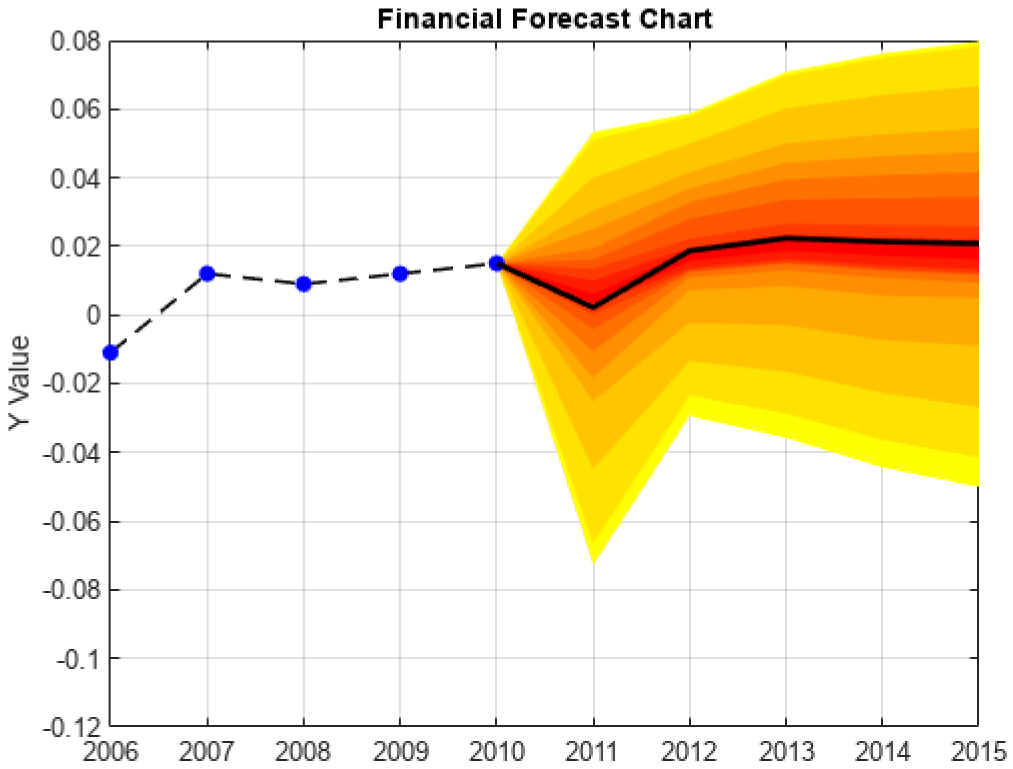
Create a Fan Plot Using Cell Array Data

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]    [0.0049]
                       [0.0533]      [0.0139]     [0.0037]     [-0.0727]     [-0.0291]   [-0.0058
               [2012]  [0.0430]      [-0.0094]    [0.0587]     [ 0.0095]     [0.0185]    [0.0205]
                       [0.0141]      [0.0337]     [0.0187]     [0.0132]      [-0.0292]   [0.0048]
               [2013]  [0.0518]      [-0.0116]    [0.0708]     [0.0112]      [0.0221]    [0.0246]
                       [0.0168]      [0.0405]     [0.0224]     [0.0157]      [-0.0356]   [0.0056]
               [2014]  [0.0546]      [-0.0171]    [0.0762]     [0.0088]      [0.0210]    [0.0239]
                       [0.0151]      [0.0419]     [0.0214]     [0.0139]      [-0.0442]   [0.0024]
               [2015]  [0.0565]      [-0.0207]    [0.0797]     [0.0072]      [0.0203]    [0.0234]
                       [0.0139]      [0.0428]     [0.0207]     [0.0126]      [-0.0499]   [0.0026]
```

Generate the fan plot.

```
fanplot (historical, forecast);
```



The dotted points are the historical lines and the filled lines indicate the mean for the forecasts. This fanplot represents a matching scenario where the time period perfectly matches for the historical and forecast data.

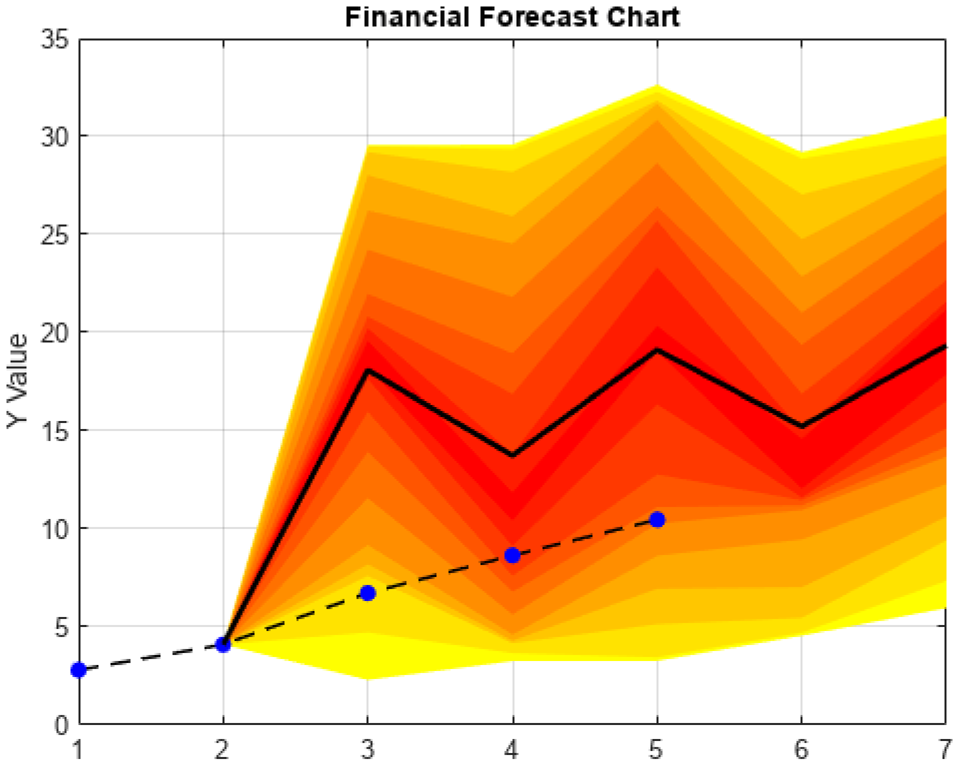
Create a Fan Plot Using Matrix Data

Define the data inputs for `historical` as a 5-by-2 matrix and `forecast` as a 5-by-21 matrix with 20 observations.

```
historical =  [ 1.0000    2.8046 ;
                2.0000    4.1040 ;
                3.0000    6.7292 ;
                4.0000    8.6486 ;
                5.0000   10.4747 ];



forecast  =   [ 3.0000   28.9874   18.3958   19.6376   29.5627    8.3462    7.1502   25.3845    2
                         20.8557   27.0691   23.0803   20.7885   18.0205   17.2294   10.0197   29

                4.0000    4.8933   27.2659    7.2206   24.4703   10.5895   15.0212   29.1137    6
                          8.0007   18.7114   19.1691   24.5963    4.2835    4.0676    3.2612   29

                5.0000   20.9732   19.7069   11.6862   25.7018   31.8940    7.2664   19.2113   10
                         31.7996    3.6419    3.2695   27.1422   10.5487   32.6529   18.8370    6

                6.0000   11.0069   29.1965    4.5551   20.2627   10.9209   15.2675   28.5359   11
                         23.9532   18.4804   25.5484    4.8747    8.0036   11.5329   11.6807   21

                7.0000    5.9699   11.1486   26.0449   13.4619   21.1196   28.8068   26.2525   10
                         21.2390   29.2396   18.4828   28.3945   21.9342   14.4642   17.2613   15
```

Generate the fan plot and return the figure handle.

```
h = fanplot(historical, forecast)
```



```
h = 
  Figure (1) with properties:

      Number: 1
        Name: ''
       Color: [1 1 1]
    Position: [360 402 560 420]
       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.

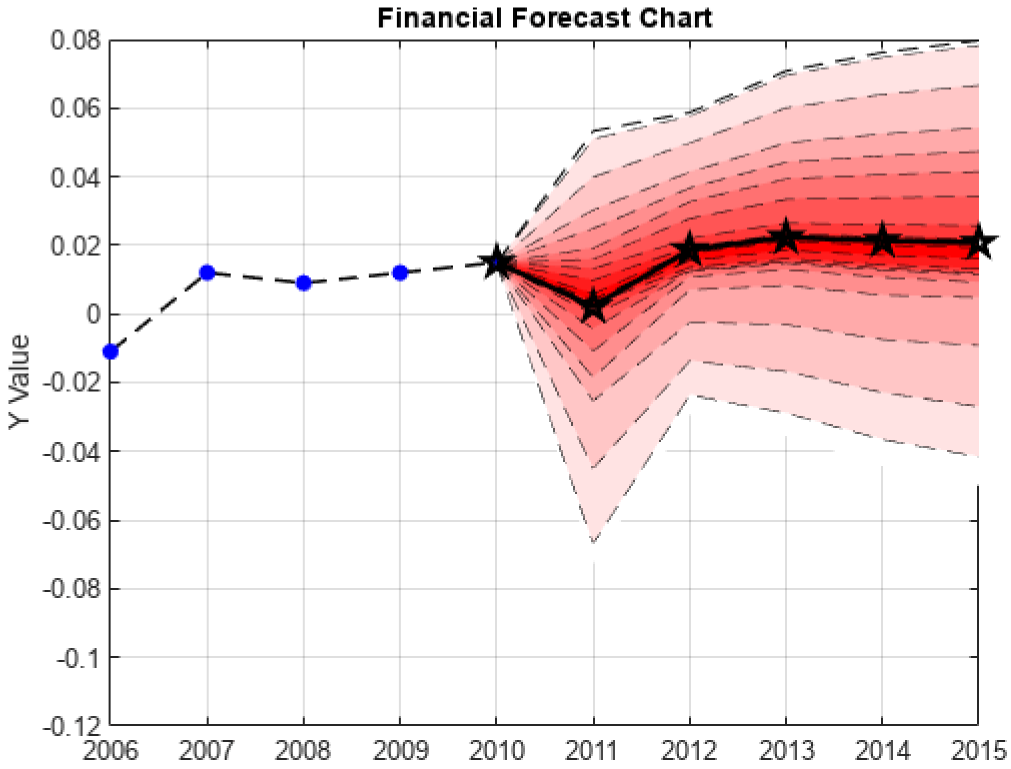
Create a Fan Plot Using Cell Array Data and Customize the Plot With Name-Value Pair Arguments

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]    [0.0049
                       [0.0533]      [0.0139]     [0.0037]     [-0.0727]     [-0.0291]   [-0.0058
               [2012]  [0.0430]      [-0.0094]    [0.0587]     [ 0.0095]     [0.0185]    [0.0205
                       [0.0141]      [0.0337]     [0.0187]     [0.0132]      [-0.0292]   [0.0048
               [2013]  [0.0518]      [-0.0116]    [0.0708]     [0.0112]      [0.0221]    [0.0246
                       [0.0168]      [0.0405]     [0.0224]     [0.0157]      [-0.0356]   [0.0056
               [2014]  [0.0546]      [-0.0171]    [0.0762]     [0.0088]      [0.0210]    [0.0239
                       [0.0151]      [0.0419]     [0.0214]     [0.0139]      [-0.0442]   [0.0024
               [2015]  [0.0565]      [-0.0207]    [0.0797]     [0.0072]      [0.0203]    [0.0234
                       [0.0139]      [0.0428]     [0.0207]     [0.0126]      [-0.0499]   [0.0026
```

Generate the fan plot using name-value pair arguments to customize the presentation.

```
fanplot(historical,forecast,'FanFaceColor',[1 1 1;1 0 0],'FanLineStyle','--','ForecastMarker','p
```



Create a Fan Plot Using Table Data

Create table of historical dates and data.

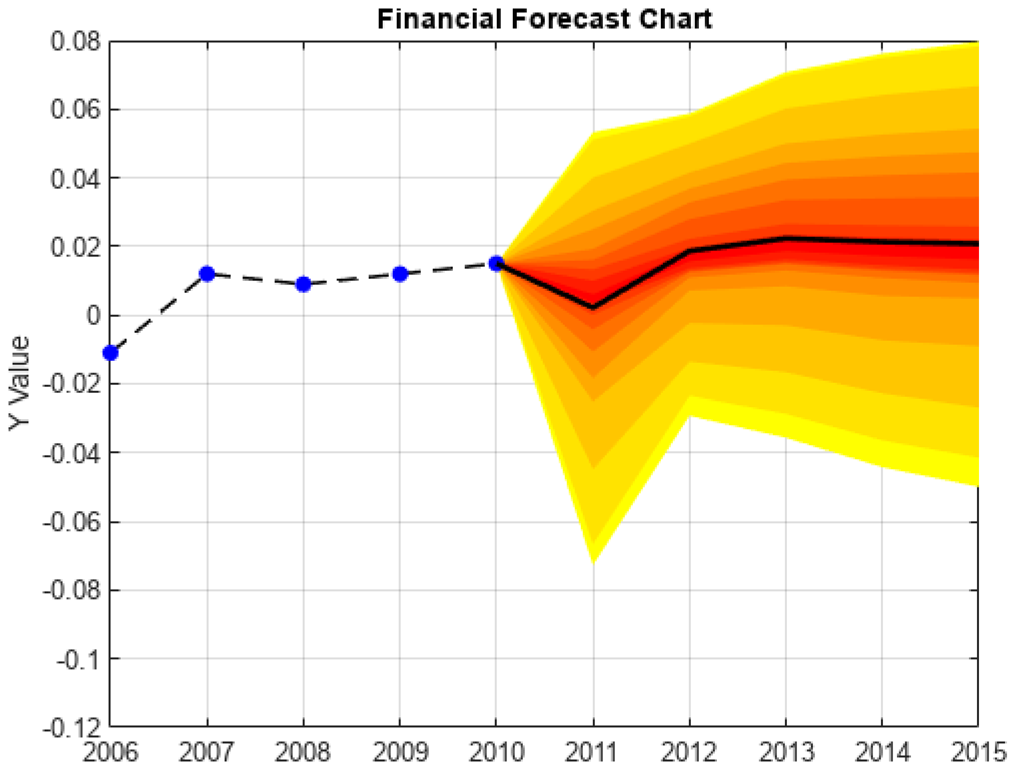
```
historicalDates = datetime(2006:2010,1,1)';
historicalData = [-0.0110;0.0120;0.0090;0.0120;0.0150];
historical = table(historicalDates,historicalData,'VariableNames',{'Dates','Data'});
```

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      0.0026
                     0.0533      0.0139     0.0037     -0.0727     -0.0291   -0.0058     0.0183
                     0.0430      -0.0094    0.0587     0.0095      0.0185    0.0205      0.0172
                     0.0141      0.0337     0.0187     0.0132      -0.0292   0.0048      0.0400
                     0.0518      -0.0116    0.0708     0.0112      0.0221    0.0246      0.0205
                     0.0168      0.0405     0.0224     0.0157      -0.0356   0.0056      0.0482
                     0.0546      -0.0171    0.0762     0.0088      0.0210    0.0239      0.0193
                     0.0151      0.0419     0.0214     0.0139      -0.0442   0.0024      0.0506
                     0.0565      -0.0207    0.0797     0.0072      0.0203    0.0234      0.0185
                     0.0139      0.0428     0.0207     0.0126      -0.0499   0.0026      0.0522
forecast = [table(forecastDates,'VariableName',{'Dates'}),array2table(forecastData)];
```

Plot the data using `fanplot`.

```
fanplot(historical,forecast);
```



Input Arguments

`historical` — Historical dates and data

matrix | cell array | table | timetable

Historical dates and data, specified as an `N`-by-2 matrix, cell array, table, or timetable 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` | `timetable`

`forecast` — Forecast dates and data

matrix | cell array of character vectors | table | timetable

Forecast dates and data, specified as an `N`-by-`M` matrix, cell array, table, or timetable 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` | `timetable`

ax — Valid axis object

object

(Optional) Valid axis object, specified as an `ax` object that is created using `axes`. The plot will be created in the axes specified by the optional `ax` argument instead of in the current axes (gca). The optional argument `ax` can precede any of the input argument combinations.

Data Types: `object`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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.

Version History

Introduced in R2014b

See Also

`bollinger` | `candle` | `highlow` | `linebreak` | `movavg` | `pointfig` | `renko` | `volarea` | `priceandvol` | `datetime` | `ceil` | `timetable`

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 datetimes or serial 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 string 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 = 1×3
```

```
      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
```

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: `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: `double`

Holiday — Holidays and nontrading-day dates
non-trading day vector is determined by the routine holidays (default) | datetime array | string array | date character vector

Holidays and nontrading-day dates, specified as vector using a datetime array, string array, date character vectors, or serial date numbers.

All dates in Holiday must be the same format: either datetimes, strings, or date character vectors. 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.

To support existing code, fbusdate also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | 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 | datetime array

Date for the first business date of a given year and month, returned as a datetime or serial date number.

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 string or date character vector, then Date is returned as a serial date number. Use the function datestr to convert serial date numbers to formatted date character vectors.
- If Holiday is a datetime array, then Date is returned as a datetime array.

.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although `fbusdate` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`busdate` | `holidays` | `isbusday` | `lbusdate` | `datetime`

Topics

"Handle and Convert Dates" on page 2-2
"Trading Calendars User Interface" on page 12-2
"UICalendar User Interface" on page 12-4

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 = datetime(2011,1,20);
Maturity = datetime(2012,1,15);
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 = [datetime(2011,1,20);
              datetime(2011,4,15);
              datetime(2011,7,15);
              datetime(2011,11,15)];
EndDates =   [datetime(2011,4,15);
              datetime(2011,7,15);
              datetime(2011,11,15);
              datetime(2012,1,15)];
ValuationDate = datetime(2011,1,20);
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 = 46.0689
```

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, the price of a bond that includes the accrued interest, 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

datetime array | string array | date character vector

Settlement date of the floating-rate bonds, specified as a scalar or a NINST-by-1 vector using a datetime array, string array, or date character vectors. If supplied as a NINST-by-1 vector of dates, all settlement dates must be the same (only a single settlement date is supported)

To support existing code, `floatdiscmargin` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Maturity` — Maturity date of the floating-rate bond
datetime array | string array | date character vector

Maturity date of the floating-rate bond, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `floatdiscmargin` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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 2-16.

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) | datetime array | string array | date character vector | serial date number

Dates for holidays, specified as NHOLIDAYS-by-1 vector using a datetime array, string array, or date character vectors. Holidays are used in computing business days.

To support existing code, floatdiscmargin also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | 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.

Version History

Introduced in R2012b

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although `floatdiscmargin` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

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

`floatmargin` | `floatbyzero` | `bndspread` | `intenvset` | `datetime`

Topics

“Fixed-Income Terminology” on page 2-15

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 = datetime(2011,1,20);
Maturity = datetime(2012,1,15);
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
```

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

datetime array | string array | date character vector

Settlement date of the floating-rate bonds, specified as a scalar or a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors. If supplied as a `NINST`-by-`1` vector of dates, all settlement dates must be the same (only a single settlement date is supported)

To support existing code, `floatmargin` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Maturity` — Maturity date of the floating-rate bond

datetime array | string array | date character vector

Maturity date of the floating-rate bond, specified as a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `floatmargin` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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 2-16.

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) | datetime array | string array | date character vector

Dates for holidays, specified as NHOLIDAYS-by-1 vector using a datetime array, string array, or date character vectors. Holidays are used in computing business days.

To support existing code, floatmargin also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | 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.

Version History

Introduced in R2012b

R2022b: Serial date numbers not recommended
Not recommended starting in R2022b

Although floatmargin supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

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

floatdiscmargin | floatbyzero | bndspread | datetime

Topics

"Fixed-Income Terminology" on page 2-15

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`

Version History

Introduced before R2006a

See Also

`cur2frac` | `cur2str`

frontier

Rolling efficient frontier

Syntax

```
[PortWts,AllMean,AllCovariance] = frontier(Universe,Window,Offset,NumPorts)
[PortWts,AllMean,AllCovariance] = frontier( ___ ,ActiveMap,Conset,NumNonNan)
```

Description

`[PortWts,AllMean,AllCovariance] = frontier(Universe,Window,Offset,NumPorts)` generates a surface of efficient frontiers showing how asset allocation influences risk and return over time.

Note An alternative for portfolio optimization is to use the `Portfolio` object 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 4-17.

`[PortWts,AllMean,AllCovariance] = frontier( ___ ,ActiveMap,Conset,NumNonNan)` specifies options using one or more optional arguments in addition to the input arguments in the previous syntax.

Input Arguments

`Universe` — Total return data for a group of securities

array

Total return data for a group of securities, specified as a number of observations (`NUMOBS`) by number of assets plus one (`NASSETS + 1`) time series array. Each row represents an observation. Column 1 contains MATLAB serial date numbers. The remaining columns contain the total return data for each security.

Data Types: `double`

`Window` — Number of data periods used to calculate each frontier

positive integer

Number of data periods used to calculate each frontier, specified as a positive integer value.

Data Types: `double`

`Offset` — Increment in number of periods between each frontier

numeric

Increment in number of periods between each frontier, specified as a numeric value.

Data Types: `double`

`NumPorts` — Number of portfolios to calculate on each frontier

positive integer

Number of portfolios to calculate on each frontier, specified as a positive integer.

Data Types: `double`

`ActiveMap` — Indicates if an asset is part of the `Universe` on the corresponding date

`NUMOBS`-by-`NASSETS` matrix of `1`'s (all assets active on all dates) (default) | matrix

(Optional) Indicates if an asset is part of the `Universe` on the corresponding date, specified as a number of observations (`NUMOBS`) by number of assets (`NASSETS`) matrix with Boolean elements corresponding to the `Universe`.

Data Types: `double`

`Conset` — Constraint matrix for a portfolio of asset investments

matrix

Constraint matrix for a portfolio of asset investments, specified using `portcons` with the `'Default'` constraint type. This single constraint matrix is applied to each frontier..

Data Types: `double`

`NumNonNan` — Minimum number of non-`NaN` points for each active asset in each window of data

`Window` - `NASSETS` (default) | numeric value

Minimum number of non-`NaN` points for each active asset in each window of data needed to perform the optimization, specified as a numeric value.

Data Types: `double`

Output Arguments

`PortWts` — Weights allocated to each asset

matrix

Weights allocated to each asset., returned as a number of curves (`NCURVES`)-by-`1` cell array, where each element is a `NPORTS`-by-`NASSETS` matrix of weights.

`AllMean` — Expected asset returns used to generate each curve on the surface

vector

Expected asset returns used to generate each curve on the surface, returned as an `NCURVES`-by-`1` cell array, where each element is a `1`-by-`NASSETS` vector of the expected asset returns.

`AllCovariance` — Covariance matrix used to generate each curve on the surface

vector

Covariance matrix used to generate each curve on the surface, returned as an `NCURVES`-by-`1` cell array, where each element is a `NASSETS`-by-`NASSETS` vector.

Version History

Introduced before R2006a

See Also

`portcons` | `portopt` | `portstats`

Topics

“Portfolio Construction Examples” on page 3-5
“Portfolio Selection and Risk Aversion” on page 3-7
“Active Returns and Tracking Error Efficient Frontier” on page 3-25
“Analyzing Portfolios” on page 3-2
“Portfolio Optimization Functions” on page 3-3

fvdisc

Future value of discounted security

Syntax

```
FutureVal = fvdisc(Settle,Maturity,Price,Discount)
FutureVal = fvdisc( ___ ,Basis)
```

Description

`FutureVal = fvdisc(Settle,Maturity,Price,Discount)` finds the amount received at maturity for a fully vested security.

`FutureVal = fvdisc( ___ ,Basis)` specifies options using an optional argument in addition to the input arguments in the previous syntax.

Examples

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(2001,2,15);
Maturity = datetime(2001,5,15);
Price = 100;
Discount = 0.0575;
Basis = 2;

FutureVal = fvdisc(Settle, Maturity, Price, Discount, Basis)
```

```
FutureVal = 101.4420
```

Input Arguments

`Settle` — Settlement date

datetime scalar | string scalar | date character vector

Settlement date, specified as a scalar datetime, string, or date character vector.

To support existing code, `fvdisc` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Maturity` — Maturity date

datetime scalar | string scalar | date character vector

Maturity date, specified as a scalar datetime, string, or date character vector.

To support existing code, `fvdisc` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Price` — Price (present value) of security

scalar numeric

Price (present value) of the security, specified as a scalar numeric.

Data Types: `double`

`Discount` — Bank discount rate of security

scalar decimal

Bank discount rate of security, specified as a scalar decimal.

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 basis of the instrument, specified as a scalar integer 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 2-16.

Data Types: `double`

Output Arguments

`FutureVal` — Amount received at maturity for fully vested security

scalar numeric

Amount received at maturity for a fully vested security, returned as a scalar numeric.

Data Types: `double`

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although `fvdisc` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Jan Mayle. *Standard Securities Calculation Methods.* Securities Industry Assn, Volumes I-II, 3rd edition, 1994

See Also

`acrudisc` | `discrate` | `prdisc` | `ylddisc` | `datetime`

Topics

“Analyzing and Computing Cash Flows” on page 2-11

fvfix

Future value with fixed periodic payments

Syntax

```
FutureVal = fvfix(Rate,NumPeriods,Payment)
FutureVal = fvfix( ___ ,PresentVal,Due)
```

Description

`FutureVal = fvfix(Rate,NumPeriods,Payment)` returns the future value of a series of equal payments.

`FutureVal = fvfix( ___ ,PresentVal,Due)` specifies options using one or more optional argument in addition to the input arguments in the previous syntax.

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
```

Input Arguments

`Rate` — Interest-rate per period

scalar numeric decimal

Interest-rate per period, specified as a scalar numeric decimal.

Data Types: `double`

`NumPeriods` — Number of payment periods

scalar numeric

Number of payment periods, specified as a scalar numeric.

Data Types: `double`

`Payment` — Payment per period

scalar numeric

Payment per period, specified as a scalar numeric.

Data Types: `double`

`PresentVal` — Present value

`0` (default) | scalar numeric

(Optional) Present value, specified as a scalar numeric.

Data Types: `double`

`Due` — When payments are due

`0` (end of period) (default) | scalar integer with value of `0` or `1`

(Optional) When payments are due, specified as a scalar integer with value of `0` (end of period) or `1` (beginning of period).

Data Types: `double`

Output Arguments

`FutureVal` — Future value of series of equal payments

scalar numeric

Future value of a series of equal payments, returned as a scalar numeric.

Version History

Introduced before R2006a

References

[1] Jan Mayle. *Standard Securities Calculation Methods.* Securities Industry Assn, Volumes I-II, 3rd edition, 1994

See Also

`fvvar` | `pvfix` | `pvvar`

Topics

“Analyzing and Computing Cash Flows” on page 2-11

fvvar

Future value of varying cash flow

Syntax

```
FutureVal = fvvar(CashFlow,Rate)
FutureVal = fvvar( ___ ,CFDates)
```

Description

`FutureVal = fvvar(CashFlow,Rate)` returns the future value of a varying cash flow.

`FutureVal = fvvar( ___ ,CFDates)` specifies options using an optional argument in addition to the input arguments in the previous syntax.

Examples

Calculate Future Value of Varying Cash Flows

This cash flow represents the yearly income from an initial investment of $10,000. The annual interest rate is 2%.

- 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], 0.02)
```

```
FutureVal = 4.7131e+03
```

An investment of $10,000 returns this irregular cash flow. The original investment and its date are included. The periodic interest rate is 3%.

- -$1000 — 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 = [datetime(2000,1,12)
           datetime(2001,2,14)
           datetime(2001,3,3)
           datetime(2001,6,14)
           datetime(2001,12,1)];

FutureVal = fvvar(CashFlow, 0.03, CFDates)
```

```
FutureVal = 1.0731e+03
```

Input Arguments

`CashFlow` — Varying cash flows

vector

Varying cash flows, specified as a vector.

Note You must include the initial investment as the initial cash flow value (a negative number).

.

Data Types: `double`

`Rate` — Periodic Interest rate

scalar numeric decimal

Periodic interest rate, specified as a scalar numeric decimal.

Data Types: `double`

`CFDates` — Dates on which the cash flows occur

assumes `CashFlow` contains regular (periodic) cash flows (default) | datetime array | string array | date character vector | serial date number

(Optional) Dates on which the cash flows occur, specified as a vector using a datetime array, string array, or date character vectors.

Note Use `CFDates` for irregular (nonperiodic) cash flows.

To support existing code, `fvvar` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

Output Arguments

`FutureVal` — Future value of a varying cash flow

scalar numeric

Future value of a varying cash flow, returned as a scalar numeric.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although fvvar supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Jan Mayle. *Standard Securities Calculation Methods.* Securities Industry Assn, Volumes I-II, 3rd edition, 1994

See Also

irr | fvfix | pvfix | pvvar | payuni | datetime

Topics

"Analyzing and Computing Cash Flows" on page 2-11

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 strings 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 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 = [datetime(2000,11,6)
              datetime(2000,12,11)
              datetime(2001,1,15)
              datetime(2001,2,5)
              datetime(2001,3,4)
              datetime(2001,4,2)
              datetime(2001,4,30)
              datetime(2001,6,25)
              datetime(2001,9,4)
              datetime(2001,11,12)];

Settle = datetime(2000,11,3);

InputCompounding = 1;
InputBasis = 2;
OutputCompounding = 1;
OutputBasis = 2;
[ZeroRates, CurveDates] = fwd2zero(ForwardRates, CurveDates,...
Settle,'InputCompounding',1,'InputBasis',2,'OutputCompounding',1,'OutputBasis',2)

ZeroRates = 10×1

    0.0469
    0.0515
    0.0531
    0.0532
    0.0538
    0.0532
    0.0536
    0.0539
    0.0556
    0.0543


CurveDates = 10x1 datetime
   06-Nov-2000
   11-Dec-2000
   15-Jan-2001
   05-Feb-2001
   04-Mar-2001
   02-Apr-2001
   30-Apr-2001
   25-Jun-2001
   04-Sep-2001
   12-Nov-2001
```

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

datetime array | string array | date character vector

Maturity dates, specified as a NUMBONDS-by-1 vector using a datetime array, string array, or date character vectors, that correspond to the ForwardRates.

To support existing code, fwd2zero also accepts serial date numbers as inputs, but they are not recommended.

Data Types: datetime | string | char

Settle — Common settlement date for ForwardRates

datetime scalar | string scalar | date character vector

Common settlement date for ForwardRates, specified as scalar datetime, string, or date character vector.

To support existing code, fwd2zero also accepts serial date numbers as inputs, but they are not recommended.

Data Types: datetime | string | char

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

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 2-16.

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 2-16.

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`

datetime | serial date number

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 strings or date character vectors, CurveDates is returned as serial date numbers. Use the function datestr to convert serial date numbers to formatted date character vectors. However, if either of the inputs for CurveDates and Settle are a datetime array, CurveDates is returned as a datetime array.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although fwd2zero supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

zero2fwd | prbyzero | pyld2zero | zbtprice | zbtyield | zero2disc | zero2fwd | zero2pyld

Topics

“Term Structure of Interest Rates” on page 2-29
“Fixed-Income Terminology” on page 2-15

geom2arith

Geometric to arithmetic moments of asset returns

Syntax

```
[ma,Ca = geom2arith(mg,Cg)
[ma,Ca = geom2arith( ___ ,t)
```

Description

`[ma,Ca = geom2arith(mg,Cg)` 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( ___ ,t)` adds an optional argument `t`.

Examples

Obtain Geometric to Arithmetic Moments of Asset Returns

This example shows several variations of using `geom2arith`.

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 the optional input `t = 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 ];
[ma, Ca] = geom2arith(m, C, 12)
```

```
ma = 4×1

    0.5508
    1.0021
    1.0906
    1.4802


Ca = 4×4

    0.0695    0.0423    0.0196         0
    0.0423    0.2832    0.1971    0.1095
    0.0196    0.1971    0.5387    0.3013
         0    0.1095    0.3013    1.0118
```

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 the optional input `t = 1/12`.

```
[ma, Ca] = geom2arith(m, C, 1/12)
```

```
ma = 4×1

    0.0038
    0.0070
    0.0076
    0.0103


Ca = 4×4

    0.0005    0.0003    0.0001         0
    0.0003    0.0020    0.0014    0.0008
    0.0001    0.0014    0.0037    0.0021
         0    0.0008    0.0021    0.0070
```

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 the optional input `t = 3`.

```
[ma, Ca] = geom2arith(m, C, 3)
```

```
ma = 4×1

    0.1377
    0.2505
    0.2726
    0.3701


Ca = 4×4

    0.0174    0.0106    0.0049         0
    0.0106    0.0708    0.0493    0.0274
    0.0049    0.0493    0.1347    0.0753
         0    0.0274    0.0753    0.2530
```

Input Arguments

`mg` — Continuously compounded or geometric mean of asset returns

vector

Continuously compounded or geometric mean of asset returns, specified as an *n*-vector.

Data Types: `double`

`Cg` — Continuously compounded or geometric covariance of asset returns

matrix

Continuously compounded or geometric covariance of asset returns, specified as an `n`-by-`n` symmetric, positive semidefinite matrix. If `Cg` is not a symmetric positive semidefinite matrix, use `nearcorr` to create a positive semidefinite matrix for a correlation matrix.

Data Types: `double`

`t` — Target period of arithmetic moments in terms of periodicity of geometric moments
`1` (default) | scalar positive numeric

(Optional) Target period of geometric moments in terms of periodicity of arithmetic moments, specified as a scalar positive numeric.

Data Types: `double`

Output Arguments

`ma` — Arithmetic mean of asset returns over the target period
vector

Arithmetic mean of asset returns over the target period (`t`), returned as an *n*-vector.

`Ca` — Arithmetic covariance of asset returns over the target period
matrix

Arithmetic covariance of asset returns over the target period (`t`), returned as an `n`-by-`n` matrix.

Algorithms

Geometric returns over period t_G are modeled as multivariate lognormal random variables with moments

$$E[\mathrm{Y}] = 1 + \mathrm{m}_G$$

and

$$\mathrm{cov}(\mathrm{Y}) = \mathrm{C}_G$$

Arithmetic returns over period t_A are modeled as multivariate normal random variables with moments

$$E[\mathrm{X}] = \mathrm{m}_A$$

$$\mathrm{cov}(\mathrm{X}) = \mathrm{C}_A$$

Given $t = t_A / t_G$, the transformation from geometric to arithmetic moments is

$$\mathrm{C}_{A_{ij}} = t\log\left(1 + \frac{\mathrm{C}_{G_{ij}}}{(1 + \mathrm{m}_{G_i})(1 + \mathrm{m}_{G_j})}\right)$$

$$\mathrm{m}_{A_i} = t\log(1 + \mathrm{m}_{G_i}) - \frac{1}{2}\mathrm{C}_{A_{ii}}$$

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`.

Version History

Introduced before R2006a

See Also

`arith2geom` | `nearcorr`

getAssetMoments

Obtain mean and covariance of asset returns from Portfolio object

Syntax

```
[AssetMean,AssetCovar] = getAssetMoments(obj)
```

Description

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 4-17.

`[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 = 4×1

    0.0042
    0.0083
    0.0100
    0.0150


assetcovar = 4×4

    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
```

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`

Data Types: `object`

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;
```

Version History

Introduced in R2011a

See Also

`setAssetMoments`

Topics

"Asset Returns and Moments of Asset Returns Using Portfolio Object" on page 4-41
"Portfolio Optimization Examples Using Financial Toolbox™" on page 4-152
"Portfolio Optimization Theory" on page 4-3

getBounds

Obtain bounds for portfolio weights from portfolio object

Syntax

```
[LowerBound,UpperBound] = getBounds(obj)
```

Description

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 4-17, “PortfolioCVaR Object Workflow” on page 5-16, and “PortfolioMAD Object Workflow” on page 6-15.

`[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 = 5×1

     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 = 5×1

     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 = 5×1

     0
     0
     0
     0
     0


UpperBound =

     []
```

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`

Data Types: `object`

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;
```

Version History

Introduced in R2011a

See Also

`setBounds`

Topics

“Working with 'Simple' Bound Constraints Using Portfolio Object” on page 4-61
“Working with 'Simple' Bound Constraints Using PortfolioCVaR Object” on page 5-54
“Working with 'Simple' Bound Constraints Using PortfolioMAD Object” on page 6-52
“Portfolio Optimization Examples Using Financial Toolbox™ ” on page 4-152
“Portfolio Set for Optimization Using Portfolio Objects” on page 4-8
“Portfolio Set for Optimization Using PortfolioCVaR Object” on page 5-8
“Portfolio Set for Optimization Using PortfolioMAD Object” on page 6-7

getBudget

Obtain budget constraint bounds from portfolio object

Syntax

```
[LowerBudget,UpperBudget] = getBudget(obj)
```

Description

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 4-17, “PortfolioCVaR Object Workflow” on page 5-16, and “PortfolioMAD Object Workflow” on page 6-15.

`[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
```

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`

Data Types: `object`

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;
```

Version History

Introduced in R2011a

See Also

`setBudget`

Topics

“Working with Budget Constraints Using Portfolio Object” on page 4-64
“Working with Budget Constraints Using PortfolioCVaR Object” on page 5-57
“Working with Budget Constraints Using PortfolioMAD Object” on page 6-55
“Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
“Portfolio Set for Optimization Using Portfolio Objects” on page 4-8
“Portfolio Set for Optimization Using PortfolioCVaR Object” on page 5-8
“Portfolio Set for Optimization Using PortfolioMAD Object” on page 6-7

getCosts

Obtain buy and sell transaction costs from portfolio object

Syntax

```
[BuyCost,SellCost] = getCosts(obj)
```

Description

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 4-17, “PortfolioCVaR Object Workflow” on page 5-16, and “PortfolioMAD Object Workflow” on page 6-15.

`[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
```

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`

Data Types: `object`

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;
```

Version History

Introduced in R2011a

See Also

`setCosts`

Topics

“Working with Transaction Costs” on page 4-53
“Working with Transaction Costs” on page 5-46
“Working with Transaction Costs” on page 6-44
“Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
“Portfolio Set for Optimization Using Portfolio Objects” on page 4-8
“Portfolio Set for Optimization Using PortfolioCVaR Object” on page 5-8
“Portfolio Set for Optimization Using PortfolioMAD Object” on page 6-7

getEquality

Obtain equality constraint arrays from portfolio object

Syntax

```
[AEquality,bEquality] = getEquality(obj)
```

Description

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 4-17, “PortfolioCVaR Object Workflow” on page 5-16, and “PortfolioMAD Object Workflow” on page 6-15.

`[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 = 1×5

     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 = 1×5

     1     1     1     0     0


bEquality = 0.5000
```

Obtain 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 = 1×5

     1     1     1     0     0


bEquality = 0.5000
```

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`

Data Types: `object`

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;
```

Version History

Introduced in R2011a

See Also

setEquality

Topics

“Working with Linear Equality Constraints Using Portfolio Object” on page 4-72
“Working with Linear Equality Constraints Using PortfolioCVaR Object” on page 5-65
“Working with Linear Equality Constraints Using PortfolioMAD Object” on page 6-63
“Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
“Portfolio Set for Optimization Using Portfolio Objects” on page 4-8
“Portfolio Set for Optimization Using PortfolioCVaR Object” on page 5-8
“Portfolio Set for Optimization Using PortfolioMAD Object” on page 6-7

getGroupRatio

Obtain group ratio constraint arrays from portfolio object

Syntax

```
[GroupA,GroupB,LowerRatio,UpperRatio] = getGroupRatio(obj)
```

Description

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 4-17, “PortfolioCVaR Object Workflow” on page 5-16, and “PortfolioMAD Object Workflow” on page 6-15.

[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 = 1×6

     1     1     1     0     0     0


GroupB = 1×6

     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 = 1×6

     1     1     1     0     0     0


GroupB = 1×6

     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 = 1×6

     1     1     1     0     0     0


GroupB = 1×6

     0     0     0     1     1     1


LowerRatio =

     []
```

```
UpperRatio = 0.5000
```

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`

Data Types: `object`

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;
```

Version History

Introduced in R2011a

See Also

`setGroupRatio`

Topics

“Working with Group Ratio Constraints Using Portfolio Object” on page 4-69
“Working with Group Constraints Using PortfolioCVaR Object” on page 5-59
“Working with Group Constraints Using PortfolioMAD Object” on page 6-57
“Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
“Portfolio Set for Optimization Using Portfolio Objects” on page 4-8
“Portfolio Set for Optimization Using PortfolioCVaR Object” on page 5-8
“Portfolio Set for Optimization Using PortfolioMAD Object” on page 6-7

getGroups

Obtain group constraint arrays from portfolio object

Syntax

```
[GroupMatrix,LowerGroup,UpperGroup] = getGroups(obj)
```

Description

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 4-17, “PortfolioCVaR Object Workflow” on page 5-16, and “PortfolioMAD Object Workflow” on page 6-15.

`[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 = 1×5

     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 = 1×5

     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 = 1×5

     1     1     1     0     0


LowerGroup =

     []

UpperGroup = 0.3000
```

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`

Data Types: `object`

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;
```

Version History

Introduced in R2011a

See Also

`setGroups`

Topics

"Working with Group Constraints Using Portfolio Object" on page 4-66
"Working with Group Constraints Using PortfolioCVaR Object" on page 5-59
"Working with Group Constraints Using PortfolioMAD Object" on page 6-57
"Portfolio Optimization Examples Using Financial Toolbox™" on page 4-152
"Portfolio Set for Optimization Using Portfolio Objects" on page 4-8
"Portfolio Set for Optimization Using PortfolioCVaR Object" on page 5-8
"Portfolio Set for Optimization Using PortfolioMAD Object" on page 6-7

getScenarios

Obtain scenarios from portfolio object

Syntax

```
Y = getScenarios(obj)
```

Description

Use the `getScenarios` function with a `PortfolioCVaR` or `PortfolioMAD` objects to obtain scenarios.

For details on the workflows, see “PortfolioCVaR Object Workflow” on page 5-16, and “PortfolioMAD Object Workflow” on page 6-15.

`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);

AssetScenarios = mvnrnd(m, C, 10);

p = PortfolioCVaR;
p = setScenarios(p, AssetScenarios);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.95);

Y = getScenarios(p)
```

```
Y = 10×4

   -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 = 10×4

   -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.

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`

Data Types: `object`

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;
```

Version History

Introduced in R2012b

See Also

`setScenarios` | `rng`

Topics

“Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-36
“Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-34

External Websites

Getting Started with Portfolio Optimization (4 min 13 sec)
CVaR Portfolio Optimization (4 min 56 sec)

getInequality

Obtain inequality constraint arrays from portfolio object

Syntax

```
[AInequality,bInequality] = getInequality(obj)
```

Description

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 4-17, “PortfolioCVaR Object Workflow” on page 5-16, and “PortfolioMAD Object Workflow” on page 6-15.

`[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 = 1×5

     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 = 1×5

     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 = 1×5

     1     1     1     0     0


bInequality = 0.5000
```

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`

Data Types: `object`

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;
```

Version History

Introduced in R2011a

See Also

`setInequality`

Topics

“Working with Linear Inequality Constraints Using Portfolio Object” on page 4-75
“Working with Linear Inequality Constraints Using PortfolioCVaR Object” on page 5-67
“Working with Linear Inequality Constraints Using PortfolioMAD Object” on page 6-65
“Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
“Portfolio Set for Optimization Using Portfolio Objects” on page 4-8
“Portfolio Set for Optimization Using PortfolioCVaR Object” on page 5-8
“Portfolio Set for Optimization Using PortfolioMAD Object” on page 6-7

getOneWayTurnover

Obtain one-way turnover constraints from portfolio object

Syntax

```
[BuyTurnover,SellTurnover] = getOneWayTurnover(obj)
```

Description

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 4-17, “PortfolioCVaR Object Workflow” on page 5-16, and “PortfolioMAD Object Workflow” on page 6-15.

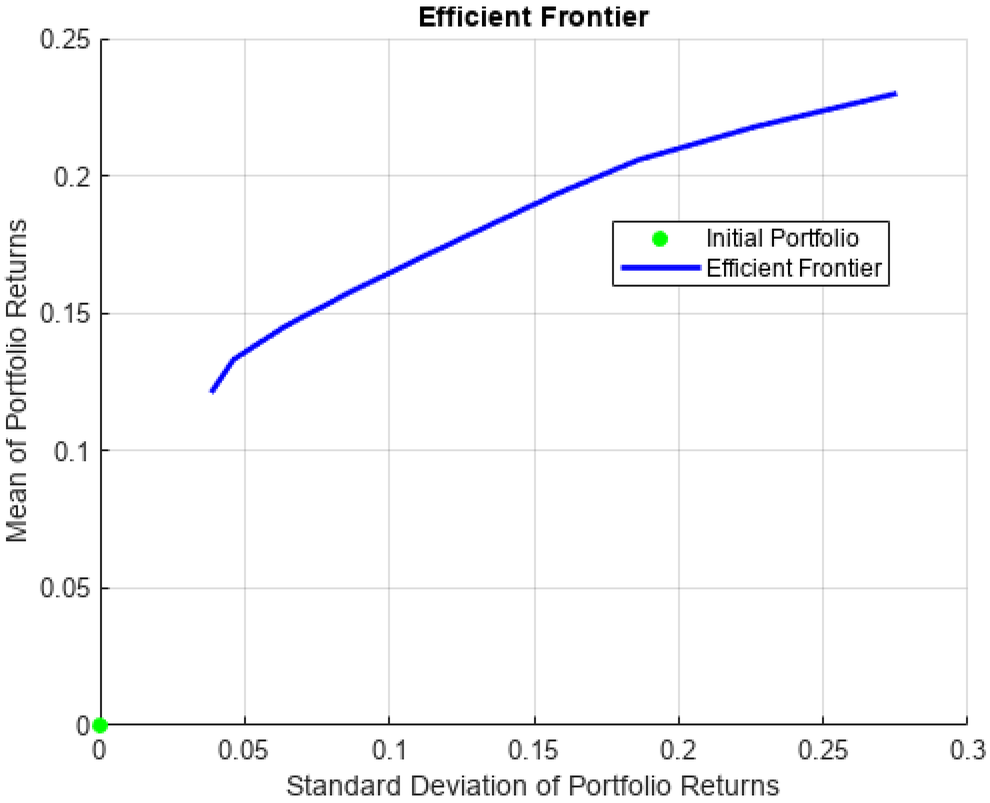
`[BuyTurnover,SellTurnover] = getOneWayTurnover(obj)` obtain one-way turnover constraints from portfolio objects.

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
```

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`

Data Types: `object`

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`).

More About

One-way Turnover Constraint

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

The constraints take the form

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

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

with

- `x` — The portfolio (*NumAssets* vector)
- x_0 — Initial portfolio (*NumAssets* vector)
- τ_B — Upper-bound for turnover constraint on purchases (scalar)
- τ_S — Upper-bound for turnover constraint on sales (scalar)

Specify one-way turnover constraints using the following properties in 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
```

Version History

Introduced in R2011a

See Also

`setOneWayTurnover` | `setTurnover`

Topics

“Working with One-Way Turnover Constraints Using Portfolio Object” on page 4-84
“Working with One-Way Turnover Constraints Using PortfolioCVaR Object” on page 5-75
“Working with One-Way Turnover Constraints Using PortfolioMAD Object” on page 6-73
“Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
“Portfolio Set for Optimization Using Portfolio Objects” on page 4-8
“Portfolio Set for Optimization Using PortfolioCVaR Object” on page 5-8
“Portfolio Set for Optimization Using PortfolioMAD Object” on page 6-7

hhigh

Highest high

Syntax

```
values = hhigh(Data)
values = hhigh( ___ ,Name,Value)
```

Description

`values = hhigh(Data)` generates a vector of highest high values from the series of high prices for the past *n* periods.

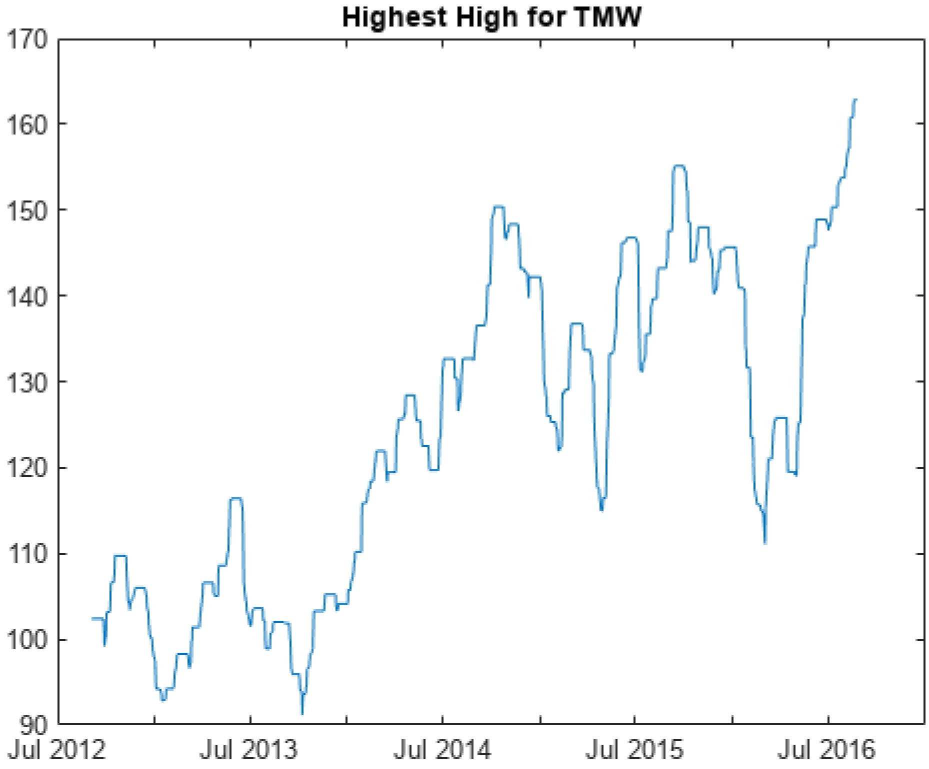
`values = hhigh( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Calculate the Highest High for a Data Series for a Stock

Load the file `SimulatedStock.mat`, which provides a timetable (`TMW`) for financial data for TMW stock.

```
load SimulatedStock.mat
values = hhigh(TMW);
plot(values.Time,values.HighestHigh)
title('Highest High for TMW')
```



Input Arguments

`Data` — Data for high prices

matrix | table | timetable

Data for high prices, specified as a matrix, table, or timetable. Timetables and tables with `M` rows must contain a variable named `'High'` (case insensitive).

Data Types: `double` | `table` | `timetable`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `values = hhigh(TMW_HIGH,'NumPeriods',10)`

`NumPeriods` — Moving window for the highest high calculation

`14` (default) | positive integer

Moving window for the highest high calculation, specified as the comma-separated pair consisting of `'NumPeriods'` and a scalar positive integer.

Data Types: `double`

Output Arguments

`values` — Highest high series
matrix | table | timetable

Highest high series, returned with the same number of rows (`M`) and the same type (matrix, table, or timetable) as the input `Data`.

Version History

Introduced before R2006a

R2023a: `fints` support removed for `Data` input argument

Behavior changed in R2023a

`fints` object support for the `Data` input argument is removed.

R2022b: Support for negative price data

Behavior changed in R2022b

The `Data` input accepts negative prices.

References

[1] Achelis, S. B. *Technical Analysis from A to Z.* Second Edition. McGraw-Hill, 1995.

See Also

`timetable` | `table` | `llow`

Topics

"Use Timetables in Finance" on page 11-7
"Convert Financial Time Series Objects (fints) to Timetables" on page 11-2

highlow

High, low, open, close chart

Note highlow is updated to accept data input as a matrix, timetable, or table.

Syntax

```
highlow(Data)
highlow(Data,Color)
h = highlow(ax, ___ )
```

Description

highlow(Data) displays a highlow chart from a series of opening, high, low, and closing prices of a security. The plots are vertical lines whose top is the high, bottom is the low, open is a left tick, and close is a right tick.

highlow(Data,Color) adds an optional argument for Color.

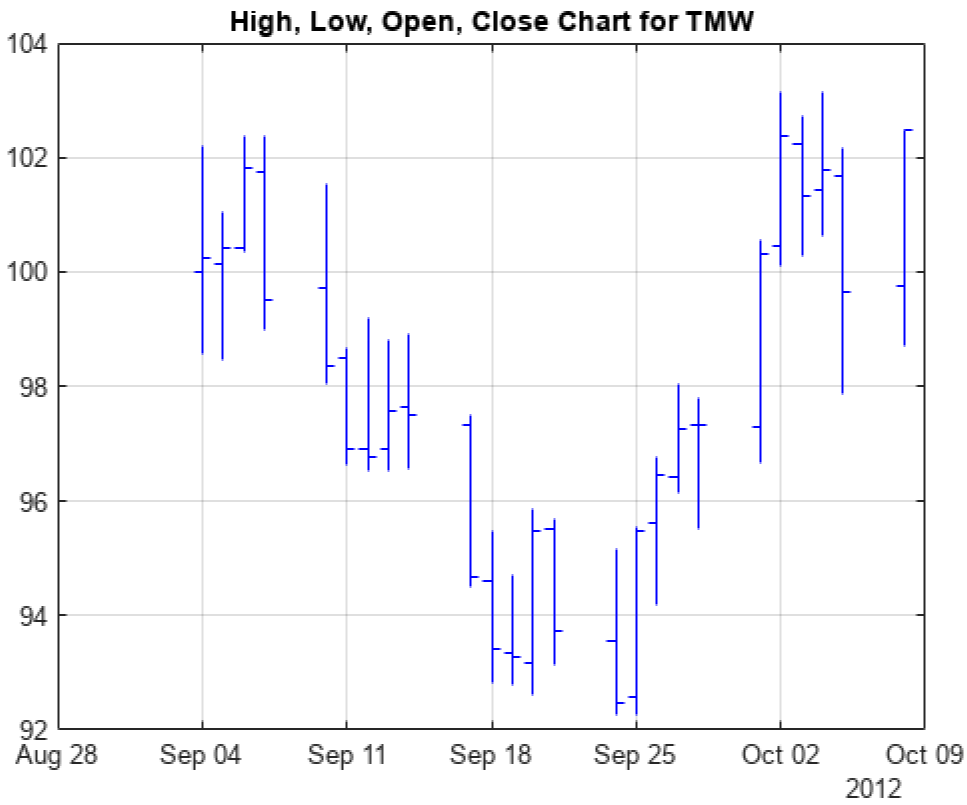
h = highlow(ax, ___) adds an optional argument for ax.

Examples

Generate a Highlow Chart for a Data Series for a Stock

Load the file SimulatedStock.mat, which provides a timetable (TMW) for financial data for TMW stock. The highlow chart plots the price data using blue lines.

```
load SimulatedStock.mat
range = 1:25;
highlow(TMW(range,:),'b');
title('High, Low, Open, Close Chart for TMW')
```



Input Arguments

`Data` — Data for opening, high, low, and closing prices

matrix | table | timetable

Data for opening, high, low, and closing prices, specified as a matrix, table, or timetable. For matrix input, `Data` is an `M`-by-4 matrix of opening, high, low, and closing prices stored in the corresponding columns. Timetables and tables with `M` rows must contain variables named `'Open'`, `'High'`, `'Low'`, and `'Close'` (case insensitive).

Data Types: `double` | `table` | `timetable`

`Color` — Three element color vector

background color of figure window (default) | color vector `[R G B]` | string

(Optional) Three element color vector, specified as a `[R G B]` color vector or a string specifying the color name. The default color differs depending on the background color of the figure window.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.

- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`"red"`	`"r"`	`[1 0 0]`	`"#FF0000"`	🟥
`"green"`	`"g"`	`[0 1 0]`	`"#00FF00"`	🟩
`"blue"`	`"b"`	`[0 0 1]`	`"#0000FF"`	🟦
`"cyan"`	`"c"`	`[0 1 1]`	`"#00FFFF"`	🟦
`"magenta"`	`"m"`	`[1 0 1]`	`"#FF00FF"`	🟪
`"yellow"`	`"y"`	`[1 1 0]`	`"#FFFF00"`	🟨
`"black"`	`"k"`	`[0 0 0]`	`"#000000"`	⬛
`"white"`	`"w"`	`[1 1 1]`	`"#FFFFFF"`	⬜

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`"#0072BD"`	🟦
`[0.8500 0.3250 0.0980]`	`"#D95319"`	🟧
`[0.9290 0.6940 0.1250]`	`"#EDB120"`	🟨
`[0.4940 0.1840 0.5560]`	`"#7E2F8E"`	🟪
`[0.4660 0.6740 0.1880]`	`"#77AC30"`	🟩
`[0.3010 0.7450 0.9330]`	`"#4DBEEE"`	🟦
`[0.6350 0.0780 0.1840]`	`"#A2142F"`	🟥

Data Types: `double` | `string`

ax — Valid axis object

current axes (`ax = gca`) (default) | axes object

(Optional) Valid axis object, specified as an axes object. The highlow plot is created in the axes specified by `ax` instead of in the current axes (`ax = gca`). The option `ax` can precede any of the input argument combinations.

Data Types: `object`

Output Arguments

h — Graphic handle of the figure

handle object

Graphic handle of the figure, returned as a handle object.

Version History

Introduced before R2006a

R2022b: Support for negative price data

The Data input accepts negative prices.

See Also

timetable | table | movavg | pointfig | kagi | linebreak | priceandvol | renko | volarea | candle

Topics

"Use Timetables in Finance" on page 11-7
"Convert Financial Time Series Objects (fints) to Timetables" on page 11-2

holdings2weights

Portfolio holdings into weights

Syntax

```
Weights = holdings2weights(Holdings,Prices)
Weights = holdings2weights( ___ ,Budget)
```

Description

`Weights = holdings2weights(Holdings,Prices)` converts portfolio holdings into portfolio weights. The weights must satisfy a budget constraint such that the weights sum to `Budget` for each portfolio.

`Weights = holdings2weights( ___ ,Budget)` converts portfolio holdings into portfolio weights using an optional argument for `Budget`.

Input Arguments

`Holdings` — Holdings value

matrix

Holdings value, returned as an `NPORTS`-by-`NASSETS` matrix that contains the holdings of `NPORTS` portfolios that contain `NASSETS` assets.

Note

- `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.)

Data Types: `double`

`Prices` — Asset prices

vector

Asset prices, specified as an `NASSETS` vector.

Data Types: `double`

`Budget` — Budget constraints

`1` (default) | scalar numeric | vector

(Optional) Budget constraints, specified as a scalar or an `NPORTS` vector of nonzero budget constraints.

Data Types: `double`

Output Arguments

`Weights` — Portfolio weights

matrix

Portfolio weights, returned as an `NPORTS`-by-`NASSETS` matrix containing the normalized weights of `NPORTS` portfolios containing `NASSETS` assets.

Data Types: `double`

Version History

Introduced before R2006a

See Also

`weights2holdings`

holidays

Holidays and nontrading days

Syntax

```
H = holidays
H = holidays(StartDate,EndDate)
H = holidays( ___ ,AltHolidays)
```

Description

`H = holidays` returns a vector or datetime array corresponding to all holidays and nontrading days.

`H = holidays(StartDate,EndDate)` returns a vector or datetime array corresponding to the holidays and nontrading days between `StartDate` and `EndDate`, inclusive.

`H = holidays( ___ ,AltHolidays)` returns a vector or datetime array 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 = 5×1

      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(2001,1,1),datetime(2001,6,23))
```

```
H = 5x1 datetime
   01-Jan-2001
   15-Jan-2001
   19-Feb-2001
   13-Apr-2001
   28-May-2001
```

Input Arguments

`StartDate` — Start date

datetime array | string array | date character vector

Start date, specified using a scalar or vector using a datetime array, string array, or date character vectors.

To support existing code, `holidays` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`EndDate` — End date

datetime array | string array | date character vector

End date, specified scalar or vector using a datetime array, string array, or date character vectors.

To support existing code, `holidays` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`AltHolidays` — Alternate list of holidays and nontrading days

datetime array | string array | date character vector

Alternate list of holidays and nontrading days, specified scalar or vector using a datetime array, string array, or date character vectors.

To support existing code, `holidays` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

Output Arguments

`H` — Dates corresponding to all holidays and nontrading days

vector | datetime

Dates corresponding to all holidays and nontrading days, returned as a vector or a datetime array of dates.

Note If `StartDate`, `EndDate`, and `AltHolidays` are all either strings or date character vectors, `H` is returned as serial date numbers. Use the function `datestr` to convert serial date numbers to formatted date character vectors. If either `StartDate`, `EndDate`, or `AltHolidays` are datetime arrays, `H` is returned as a datetime array.

More About

holidays

The holidays function is based on a modern five-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 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'.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although holidays supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

busdate | createholidays | isbusday | lbusdate | nyseclosures | datetime

Topics

“Handle and Convert Dates” on page 2-2
“Trading Calendars User Interface” on page 12-2
“UICalendar User Interface” on page 12-4

inforatio

Calculate information ratio for one or more assets

Syntax

```
inforatio(Asset,Benchmark)
[Ratio,TE] = inforatio(Asset,Benchmark)
```

Description

`inforatio(Asset,Benchmark)` computes the information ratio for each asset relative to the `Benchmark`.

`[Ratio,TE] = inforatio(Asset,Benchmark)` computes the information ratio and tracking error for each asset relative to the `Benchmark`.

Examples

Compute Information Ratio

This example show how to calculate the information ratio using `inforatio` with example data, where the mean return of the market series is used as the return of the benchmark.

You can use inforatio to compute the information ratio for the given asset return data and the riskless asset return.

```
load FundMarketCash
Returns = tick2ret(TestData);
Benchmark = Returns(:,2);
InfoRatio = inforatio(Returns, Benchmark)
```

```
InfoRatio = 1×3

    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®.

Use Information Ratio to Compute Tracking Error

This example show how to calculate the tracking error using `inforatio` with example data, where the mean return of the market series is used as the return of the benchmark.

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.

```
load FundMarketCash
Returns = tick2ret(TestData);
Benchmark = Returns(:,2);
[InfoRatio, TrackingError] = inforatio(Returns, Benchmark)
```

```
InfoRatio = 1×3

    0.0432       NaN   -0.0315


TrackingError = 1×3

    0.0187         0    0.0390
```

Tracking error, also know as active risk, measures the volatility of active returns. 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.

Input Arguments

Asset — Asset returns

matrix

Asset returns, specified as a `NUMSAMPLES x NUMSERIES` matrix with `NUMSAMPLES` observations of asset returns for `NUMSERIES` asset return series.

Data Types: `double`

Benchmark — Returns for a benchmark asset

vector

Returns for a benchmark asset, specified as a `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.

Data Types: `double`

Output Arguments

Ratio — Information ratios

vector

Information ratios, returned as 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 — Tracking errors

vector

Tracking errors, returned as 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.

Version History

Introduced in R2006b

References

[1] Grinold, R. C. and Ronald N. Kahn. *Active Portfolio Management.* 2nd. Edition. McGraw-Hill, 2000.

[2] Treynor, J. 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

“Performance Metrics Overview” on page 7-2

irr

Internal rate of return

Syntax

```
Return = irr(CashFlow)
[Return,AllRates] = irr( ___ )
```

Description

`Return = irr(CashFlow)` calculates the internal rate of return 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).

`[Return,AllRates] = irr( ___ )` calculates the internal rate of return and a vector of all internal rates for a series of periodic cash flows.

Examples

Find a Single and Multiple Internal Rates of Return

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])
```

```
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 Multiple Internal Rates 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]
```

```
CashFlow = 1×4

       -1000        6000      -10900        5800
```

```
[Return, AllRates] = irr(CashFlow)
```

```
Return = 1.0000
```

```
AllRates = 3×1

   -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)
```

```
PV = -196.0932
```

The second argument is the 10% market rate. The present value is -196.0932, negative, so the project is undesirable.

Input Arguments

`CashFlow` — Stream of periodic cash flows
vector | matrix

Stream of periodic cash flows, specified as a vector or matrix. 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.

Data Types: `double`

Output Arguments

`Return` — Internal rate of return
vector

Internal rate of return associated to `CashFlow`, returned as a vector whose entry `j` is an internal rate of return for column `j` in `CashFlow`

`AllRates` — All the internal rates of return
serial date number | matrix

All the internal rates of return associated with CashFlow, returned as 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).

Version History

Introduced before R2006a

References

[1] Brealey and Myers. *Principles of Corporate Finance.* McGraw-Hill Higher Education, Chapter 5, 2003.

[2] Hazen G. “A New Perspective on Multiple Internal Rates of Return.” *The Engineering Economist.* Vol. 48-1, 2003, pp. 31–51.

See Also

effrr | mirr | nomrr | xirr | pvvar

Topics

“Interest Rates/Rates of Return” on page 2-11
“Analyzing and Computing Cash Flows” on page 2-11
“Performance Metrics Overview” on page 7-2

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 = [datetime(2001,2,15); datetime(2001,2,16) ; datetime(2001,2,17)];
Busday = isbusday(Date)
```

```
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(datetime(2003,6,21), [], 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.

Input Arguments

`Date` — Date being checked

datetime array | string array | date character vector

Date being checked, specified as a scalar or a vector using a datetime array, string array, or date character vectors. `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.

To support existing code, `isbusday` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Holiday` — Holidays and nontrading-day dates

non-trading day vector is determined by the routine `holidays` (default) | datetime array | string array | date character vector

Holidays and nontrading-day dates, specified as a vector using a datetime array, string array, or date character vectors.

All dates in `Holiday` must be the same format: either datetimes, strings, date character vectors, or serial date numbers. The `holidays` function supplies the default vector.

To support existing code, `isbusday` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `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.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although `isbusday` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`busdate` | `fbusdate` | `holidays` | `lbusdate` | `datetime`

Topics

"Handle and Convert Dates" on page 2-2
"Trading Calendars User Interface" on page 12-2
"UICalendar User Interface" on page 12-4

kagi

Kagi chart

Note kagi is updated to accept data input as a matrix, timetable, or table.

The syntax for kagi has changed. Previously, when using table input, the first column of dates could be serial date numbers, date character vectors, or datetime arrays, and you were required to have specific number of columns.

When using table input, the new syntax for kagi supports:

- No need for time information. If you want to pass in date information, use timetable input.
- No requirement of specific number of columns. However, you must provide valid column names. kagi must contain a column named ‘price’ (case insensitive).

Syntax

```
kagi(Data)
h = kagi(ax,Data)
```

Description

kagi(Data) plots a Kagi chart from a series of prices.

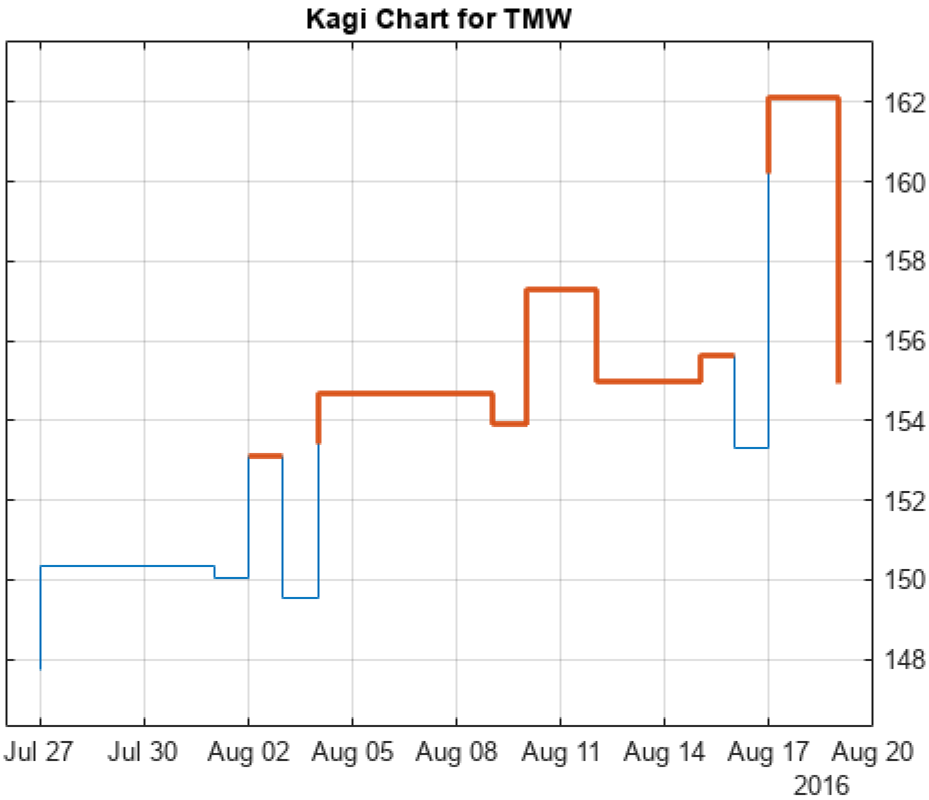
h = kagi(ax,Data) adds an optional argument for ax.

Examples

Generate a Kagi Chart for a Data Series for a Stock

Load the file SimulatedStock.mat, which provides a timetable (TMW) for financial data for TMW stock. This Kagi chart is for closing prices of the stock TMW for the most recent 21 days. Note that the variable name of asset price is be renamed to 'Price' (case insensitive).

```
load SimulatedStock.mat
TMW.Properties.VariableNames{'Close'} = 'Price';
kagi(TMW(end-20:end,:))
title('Kagi Chart for TMW')
```



Input Arguments

`Data` — Data for a series of prices

matrix | table | timetable

Data for a series of prices, specified as a matrix, table, or timetable. Timetables and tables with `M` rows must contain a variable named `'Price'` (case insensitive).

Data Types: `double` | `table` | `timetable`

ax — Valid axis object

current axes (`ax = gca`) (default) | axes object

(Optional) Valid axis object, specified as an axes object. The kagi plot is created in the axes specified by `ax` instead of in the current axes (`ax = gca`). The option `ax` can precede any of the input argument combinations.

Data Types: `object`

Output Arguments

h — Graphic handle of the figure

handle object

Graphic handle of the figure, returned as a handle object.

Version History

Introduced in R2008a

R2022b: Support for negative price data
Behavior changed in R2022b

The `Data` input accepts negative prices.

See Also

`timetable` | `table` | `movavg` | `pointfig` | `highlow` | `linebreak` | `priceandvol` | `renko` | `volarea` | `candle`

Topics

"Use Timetables in Finance" on page 11-7
"Convert Financial Time Series Objects (fints) to Timetables" on page 11-2

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 serial 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 string 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 = lbusdate(2003, 5, [], Weekend,'datetime')
```

```
Date = datetime
   31-May-2003
```

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: 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: double

Holiday — Holidays and nontrading-day dates

non-trading day vector is determined by the routine holidays (default) | datetime array | string array | date character vector

Holidays and nontrading-day dates, specified as a vector using a datetime array, string array, or date character vectors.

All dates in Holiday must be the same format: either datetimes, strings, or date character vectors. 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.

To support existing code, lbusdate also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | 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

datetime | serial date number

Date for the last business date of a given year and month, returned as a datetime or serial date number.

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 string or date character vector, then Date is returned as a serial date number. Use the function datestr to convert serial date numbers to formatted date character vectors.
- If Holiday is a datetime array, then Date is returned as a datetime array.

.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although lbusdate supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

busdate | fbusdate | holidays | isbusday | datetime

Topics

“Handle and Convert Dates” on page 2-2

“Trading Calendars User Interface” on page 12-2
“UICalendar User Interface” on page 12-4

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,:))
```

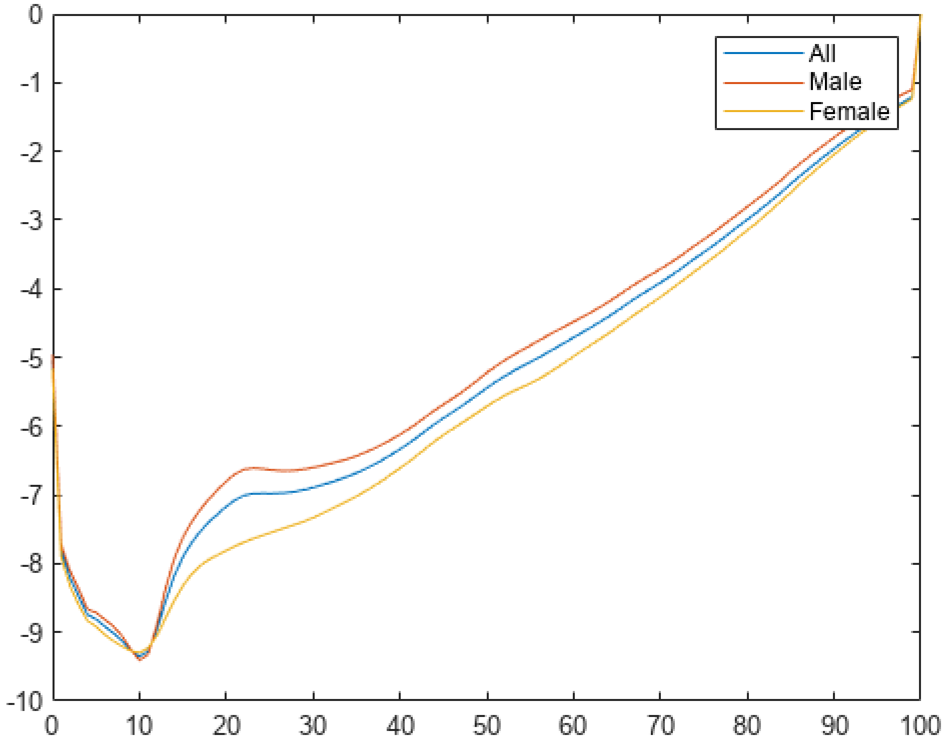
```
  1.0e+05 *

   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
   40.4062   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   32.2812
   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 = 8×3

    0.0005    0.0006    0.0004
    0.0592    0.0819    0.0192
    0.1452    0.1626    0.1048
    0.0007    0.0011    0.0007
    6.2853    6.7641    1.1037
   24.1386   24.2894   53.1848
    0.0000    0.0000    0.0000
    1.0971    1.0987    1.1100
```

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
```

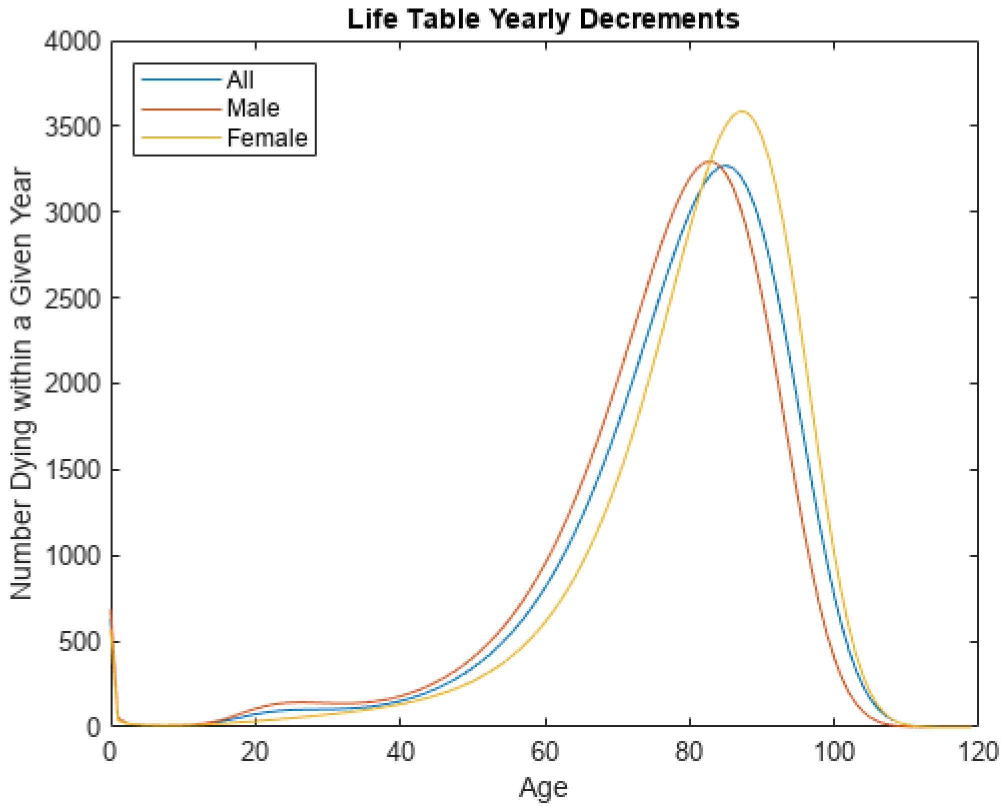
Convert life table series into life tables with forced termination.

```
[~,~,dx] = lifetableconv((0:120),qx,'qx');
display(dx(1:20,:))
```

```
  630.9935  686.9471  571.6100
   48.7919   55.1033   40.9870
   24.8017   26.3778   23.6166
   17.0833   17.5877   17.0317
   13.6183   13.8188   13.6142
   11.8664   12.0076   11.6314
   10.9785   11.1573   10.4905
   10.5999   10.8605    9.9488
   10.5760   10.9396    9.8952
   10.8792   11.3613   10.2718
   11.6084   12.2508   11.0419
   12.9918   13.9270   12.1764
   15.3470   16.8833   13.6482
   18.9922   21.6788   15.4301
   24.1370   28.7659   17.4943
   30.7981   38.3208   19.8133
   38.7691   50.1484   22.3602
   47.6516   63.6906   25.1097
   56.9291   78.1271   28.0384
   66.0577   92.5264   31.1256
```

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');
```



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 15-1059.
- 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 15-1060.

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 15-1059.

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 15-1059.

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 15-1059.

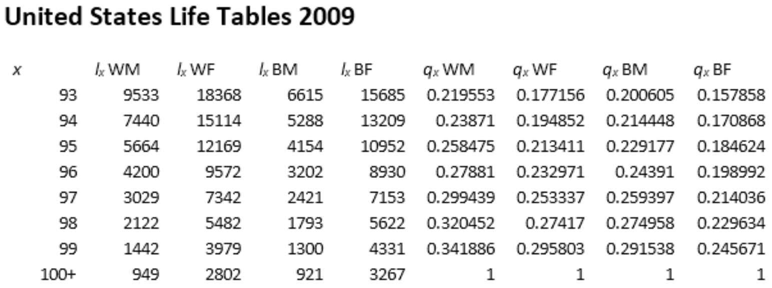
The series dx is the number of people who die out of 100,000 alive at birth, between age *x* and the next age in the series.

More About

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

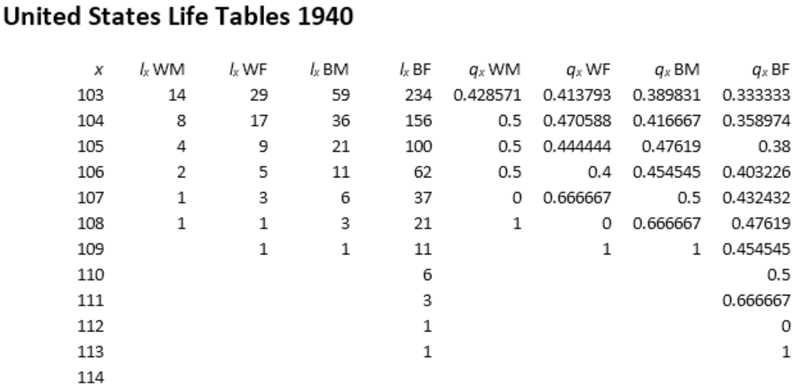
x	l_x WM	l_x WF	l_x BM	l_x BF	q_x WM	q_x WF	q_x BM	q_x BF
93	9533	18368	6615	15685	0.219553	0.177156	0.200605	0.157858
94	7440	15114	5288	13209	0.23871	0.194852	0.214448	0.170868
95	5664	12169	4154	10952	0.258475	0.213411	0.229177	0.184624
96	4200	9572	3202	8930	0.27881	0.232971	0.24391	0.198992
97	3029	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.



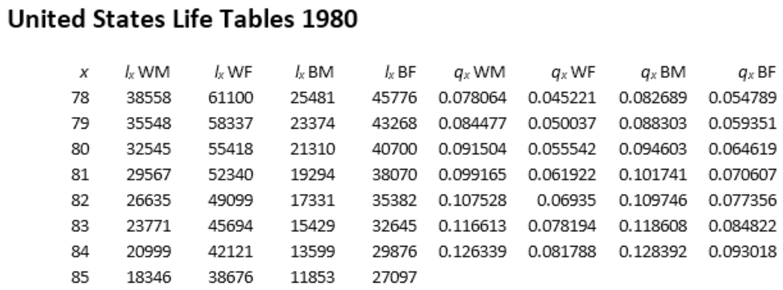
United States Life Tables 1940

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.

Version History

Introduced in R2015a

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 2-46
"About Life Tables" on page 2-44

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)
```

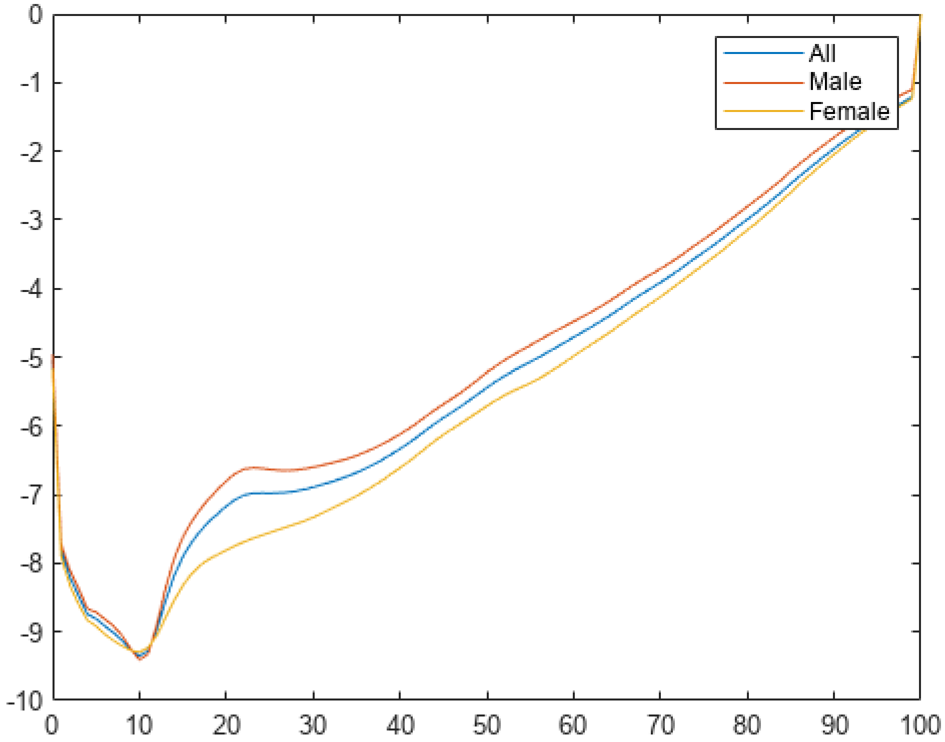
```
a = 8×3

    0.0005    0.0006    0.0004
    0.0592    0.0819    0.0192
    0.1452    0.1626    0.1048
    0.0007    0.0011    0.0007
    6.2853    6.7641    1.1037
   24.1386   24.2894   53.1848
    0.0000    0.0000    0.0000
    1.0971    1.0987    1.1100
```

```
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)
```



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 $\widehat{q}_x$ for $x = 1, \ldots, N$, the first objective (which is the preferred objective) has the form

  $$\Phi = \sum_{x=1}^{N} \left(1 - \frac{\widehat{q}_x}{q_x}\right)^2$$
- `'logratio'` — Given raw data q_x and model estimates $\widehat{q}_x$ for $x = 1, \ldots, N$, the second objective has the form

  $$\Phi = \sum_{x=1}^{N} \left(\log(\widehat{q}_x) - \log(q_x)\right)^2$$

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.

Version History

Introduced in R2015a

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 2-46
“About Life Tables” on page 2-44

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 = 8×3

    0.0005    0.0006    0.0004
    0.0592    0.0819    0.0192
    0.1452    0.1626    0.1048
    0.0007    0.0011    0.0007
    6.2853    6.7641    1.1037
   24.1386   24.2894   53.1848
    0.0000    0.0000    0.0000
    1.0971    1.0987    1.1100
```

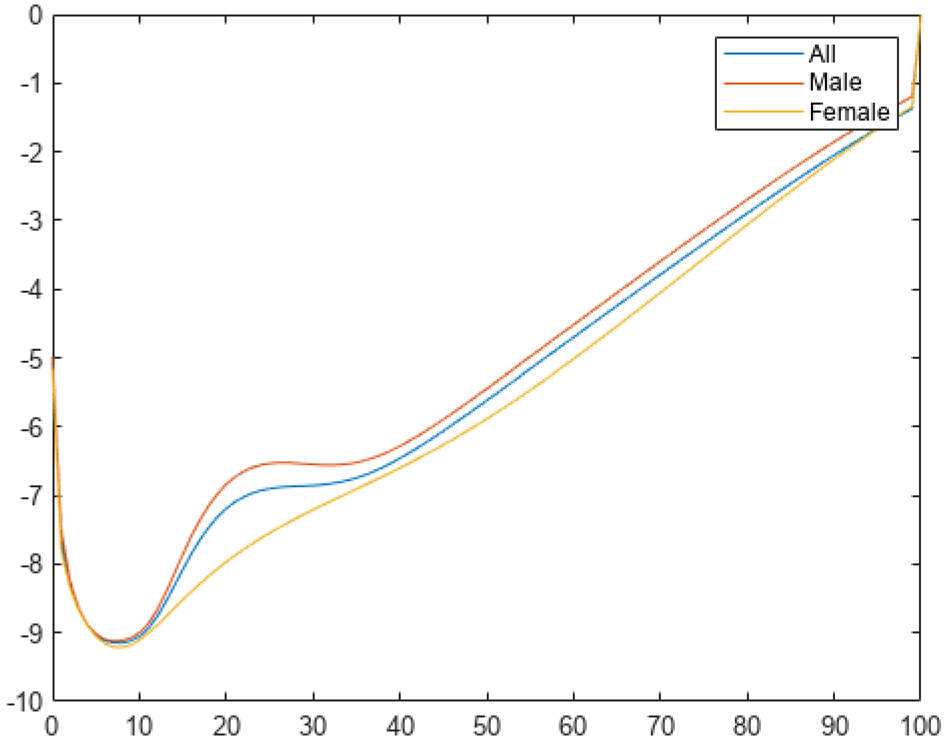
Generate a life table series from the calibrated mortality model.

```
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 = 8×3

    0.0005    0.0006    0.0004
    0.0592    0.0819    0.0192
    0.1452    0.1626    0.1048
    0.0007    0.0011    0.0007
    6.2855    6.7638    1.1158
   24.1384   24.2895   52.6087
    0.0000    0.0000    0.0000
    1.0971    1.0987    1.1099
```

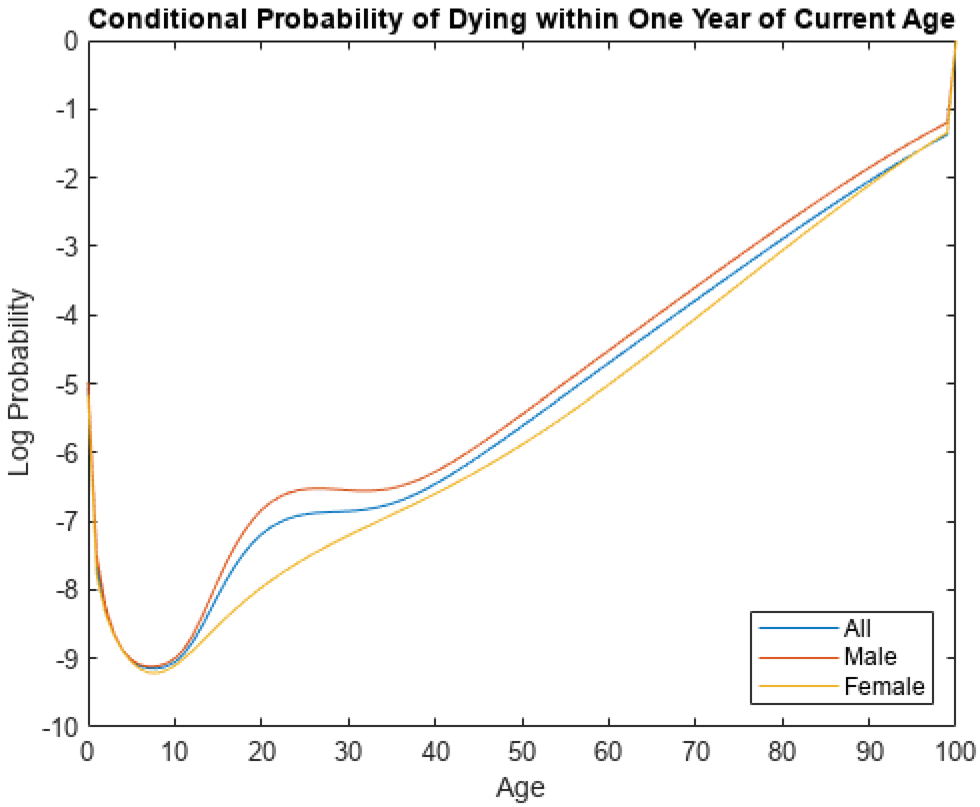
Generate a life table series from the calibrated mortality model.

```
qx = lifetablegen((0:100), a)
```

```
qx = 101×3

    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');
```



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 15-1071). 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.

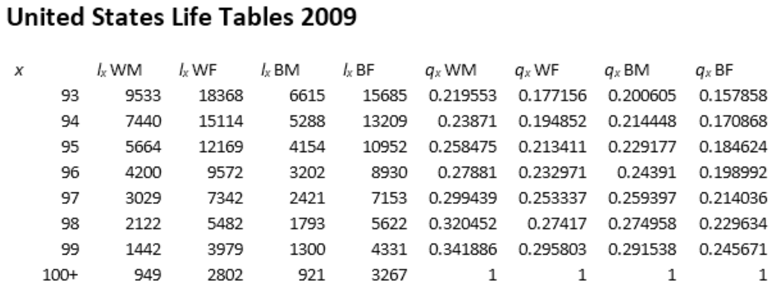
The last row of the `N`-by-`num` output for `dx` are values for all ages on or after the last age in *x* (due to "Forced Termination" on page 15-1071). Therefore, the last row of `dx` contains the remaining count of 100,000 people alive at birth who have not died by the last age.

More About

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.

Version History

Introduced in R2015a

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

`lifetablefit` | `lifetableconv`

Topics

“Case Study for Life Tables Analysis” on page 2-46
“About Life Tables” on page 2-44

linebreak

Line break chart

Note `linebreak` is updated to accept data input as a matrix, `timetable`, or `table`.

The syntax for `linebreak` has changed. Previously, when using table input, the first column of dates could be serial date numbers, date character vectors, or datetime arrays, and you were required to have specific number of columns.

When using table input, the new syntax for `linebreak` supports:

- No need for time information. If you want to pass in date information, use timetable input.
- No requirement of specific number of columns. However, you must provide valid column names. `linebreak` must contain a column named `‘price’` (case insensitive).

Syntax

```
linebreak(Data)
h = linebreak(ax,Data)
```

Description

`linebreak(Data)` plots the asset data, in a line break chart.

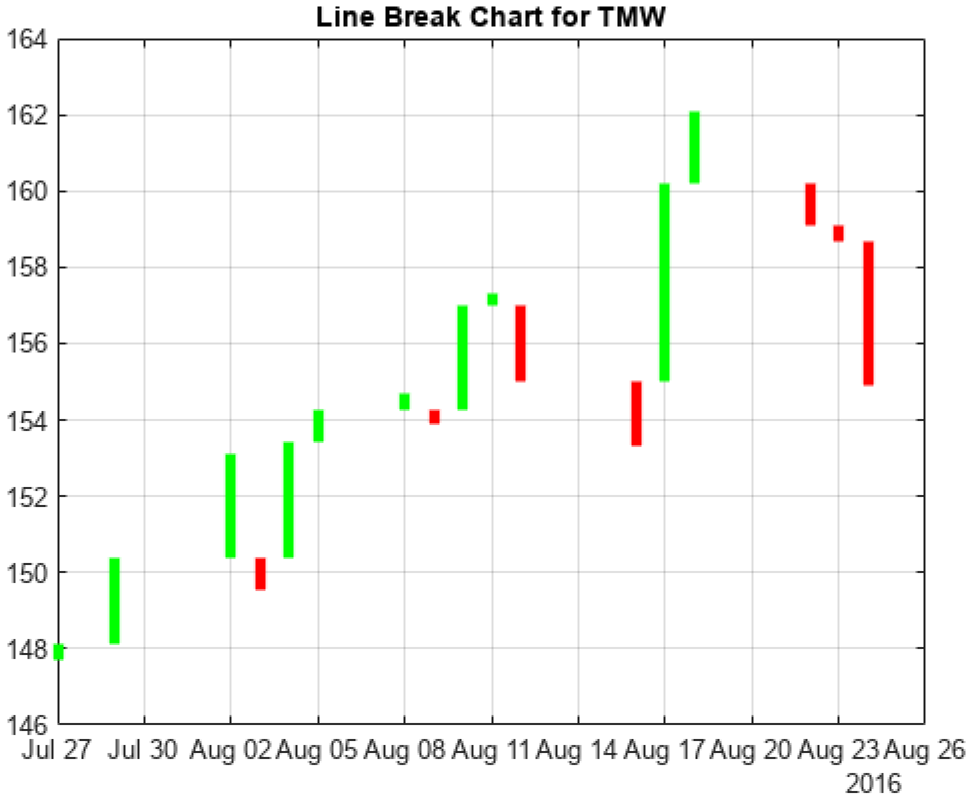
`h = linebreak(ax,Data)` adds an optional argument for `ax`.

Examples

Generate a Line Break Chart for a Data Series for a Stock

Load the file `SimulatedStock.mat`, which provides a timetable (`TMW`) for financial data for TMW stock. This Linebreak chart is for closing prices of the stock TMW for the most recent 21 days. Note that the variable name of asset price is be renamed to `'Price'` (case insensitive).

```
load SimulatedStock.mat
TMW.Properties.VariableNames{'Close'} = 'Price';
linebreak(TMW(end-20:end,:))
title('Line Break Chart for TMW')
```



Input Arguments

`Data` — Data for a series of prices

matrix | table | timetable

Data for a series of prices, specified as a matrix, table, or timetable. Timetables and tables with `M` rows must contain a variable named `'Price'` (case insensitive).

Data Types: `double` | `table` | `timetable`

ax — Valid axis object

current axes (`ax = gca`) (default) | axes object

(Optional) Valid axis object, specified as an axes object. The linebreak plot is created in the axes specified by `ax` instead of in the current axes (`ax = gca`). The option `ax` can precede any of the input argument combinations.

Data Types: `object`

Output Arguments

h — Graphic handle of the figure

handle object

Graphic handle of the figure, returned as a handle object.

Version History

Introduced in R2008a

R2022b: Support for negative price data

Behavior changed in R2022b

The `Data` input accepts negative prices.

See Also

`timetable` | `table` | `movavg` | `pointfig` | `highlow` | `kagi` | `priceandvol` | `renko` | `volarea` | `candle`

Topics

"Use Timetables in Finance" on page 11-7
"Convert Financial Time Series Objects (fints) to Timetables" on page 11-2

llow

Lowest low

Syntax

```
values = llow(Data)
values = llow( ___ ,Name,Value)
```

Description

`values = llow(Data)` generates a vector of lowest low values from the series of low prices for the past *n* periods.

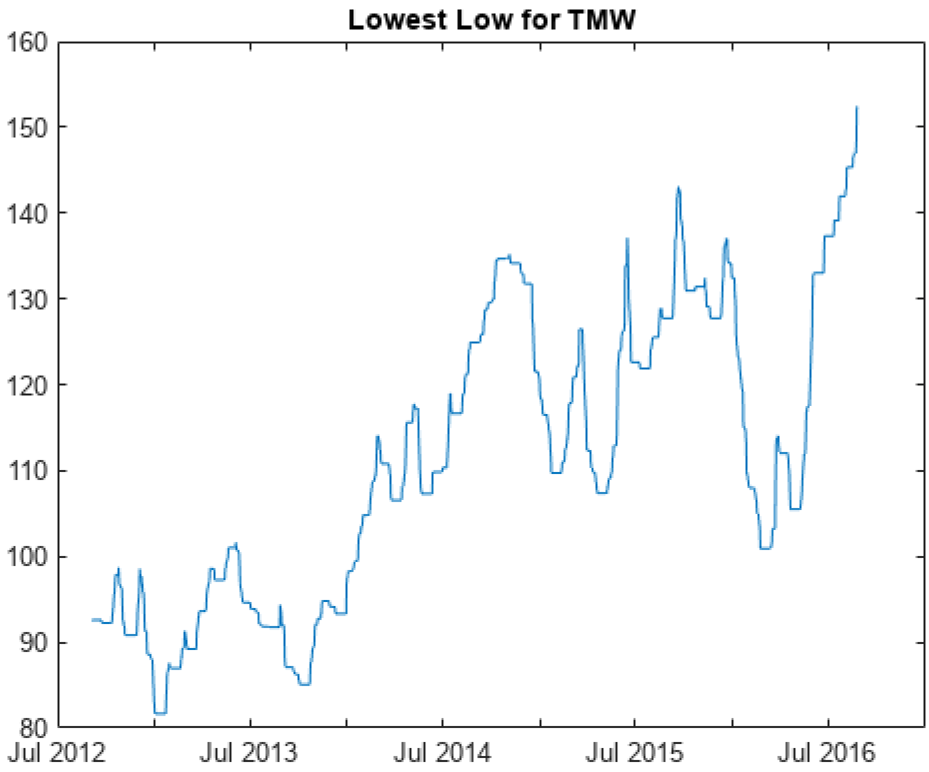
`values = llow( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Calculate the Lowest Low for a Data Series for a Stock

Load the file `SimulatedStock.mat`, which provides a timetable (`TMW`) for financial data for TMW stock.

```
load SimulatedStock.mat
values = llow(TMW);
plot(values.Time,values.LowestLow)
title('Lowest Low for TMW')
```



Input Arguments

`Data` — Data for low prices

matrix | table | timetable

Data for low prices, specified as a matrix, table, or timetable. Timetables and tables with `M` rows must contain a variable named `'Low'` (case insensitive).

Data Types: `double` | `table` | `timetable`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `values = llow(TMW_LOW,'NumPeriods',10)`

`NumPeriods` — Moving window for the lowest low calculation

`14` (default) | positive integer

Moving window for the lowest low calculation, specified as the comma-separated pair consisting of `'NumPeriods'` and a scalar positive integer.

Data Types: `double`

Output Arguments

`values` — Lowest low series
matrix | table | timetable

Lowest low series, returned with the same number of rows (`M`) and the same type (matrix, table, or timetable) as the input `Data`.

Version History

Introduced before R2006a

R2023a: `fints` support removed for `Data` input argument
Behavior changed in R2023a

`fints` object support for the `Data` input argument is removed.

R2022b: Support for negative price data
Behavior changed in R2022b

The `Data` input accepts negative prices.

References

[1] Achelis, S. B. *Technical Analysis from A to Z.* Second Edition. McGraw-Hill, 1995.

See Also

`timetable` | `table` | `hhigh`

Topics
“Use Timetables in Finance” on page 11-7
“Convert Financial Time Series Objects (fints) to Timetables” on page 11-2

lpm

Compute sample lower partial moments of data

Syntax

```
lpm(Data)
lpm(Data,MAR,Order)
Moment = lpm(Data,MAR,Order)
```

Description

`lpm(Data)` computes lower partial moments for asset returns `Data` relative to a default value for `MAR` for each asset in a `NUMORDERS x NUMSERIES` matrix and a default value for `Order`.

`lpm(Data,MAR,Order)` computes lower partial moments for asset returns `Data` relative to `MAR` for each asset in a `NUMORDERS x NUMSERIES` matrix.

`Moment = lpm(Data,MAR,Order)` computes lower partial moments for asset returns `Data` relative to `MAR` for each asset in a `NUMORDERS x NUMSERIES` matrix `Moment`.

Examples

Compute Lower Partial Moments

This example shows how 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
```

```
Assets = 1x3 cell
    {'Fund'}    {'Market'}    {'Cash'}
```

```
MAR = mean(Returns(:,3))
```

```
MAR = 0.0017
```

```
LPM = lpm(Returns, MAR, [0 1 2])
```

```
LPM = 3×3

    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.

Input Arguments

Data — Asset returns

matrix

Asset returns, specified as a NUMSAMPLES-by-NUMSERIES matrix with NUMSAMPLES observations of NUMSERIES asset returns.

Data Types: double

MAR — Minimum acceptable return

0 (default) | numeric

(Optional) Minimum acceptable return, specified as a scalar numeric. MAR is a cutoff level of return such that all returns above MAR contribute nothing to the lower partial moment.

Data Types: double

Order — Moment orders

0 (default) | scalar numeric | vector

(Optional) Moment orders, specified as a either a scalar or a NUMORDERS vector of nonnegative integer moment orders. If no order specified, the default Order = 0, which is the shortfall probability. Although the lpm function works for noninteger orders and, in some cases, for negative orders, this falls outside customary usage.

Data Types: double

Output Arguments

Moment — Lower partial moments

matrix

Lower partial moments, returned as a NUMORDERS x NUMSERIES matrix of lower partial moments with NUMORDERS Orders and NUMSERIES series, that is, each row contains lower partial moments for a given order.

Note To compute upper partial moments, reverse the signs of both Data and MAR (do not reverse the sign of the output). The lpm 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))`

More About

Lower Partial Moments

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.

Version History

Introduced in R2006b

References

[1] Bawa, V.S. "Safety-First, Stochastic Dominance, and Optimal Portfolio Choice." *Journal of Financial and Quantitative Analysis*. Vol. 13, No. 2, June 1978, pp. 255–271.

[2] Harlow, W.V. "Asset Allocation in a Downside-Risk Framework." *Financial Analysts Journal*. Vol. 47, No. 5, September/October 1991, pp. 28–40.

[3] Harlow, W.V. 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.

[4] Sortino, F.A. and Robert van der Meer. "Downside Risk." *Journal of Portfolio Management*. Vol. 17, No. 5, Spring 1991, pp. 27–31.

See Also

`elpm`

Topics

“Performance Metrics Overview” on page 7-2

macd

Moving Average Convergence/Divergence (MACD)

Syntax

`[MACDLine,SignalLine] = macd(Data)`

Description

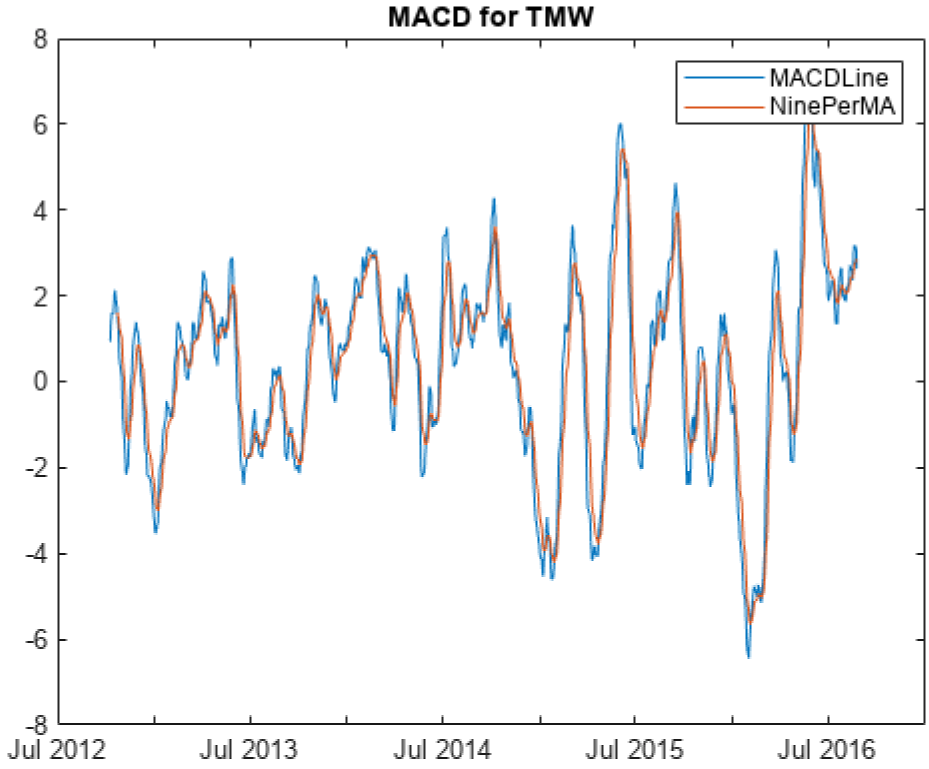
`[MACDLine,SignalLine] = macd(Data)` calculates the Moving Average Convergence/Divergence (MACD) line from the series of data and the nine-period exponential moving average from the `MACDLine`.

Examples

Calculate the Moving Average Convergence/Divergence for a Stock

Load the file `SimulatedStock.mat`, which provides a timetable (`TMW`) for financial data for TMW stock.

```
load SimulatedStock.mat
[MACDLine, signalLine]= macd(TMW);
plot(MACDLine.Time,MACDLine.Close,signalLine.Time,signalLine.Close);
legend('MACDLine','NinePerMA')
title('MACD for TMW')
```



Input Arguments

`Data` — Data with high, low, open, close information

matrix | table | timetable

Data with high, low, open, close information, specified as a matrix, table, or timetable. For matrix input, `Data` is an `M`-by-4 matrix of high, low, opening, and closing prices. Timetables and tables with `M` rows must contain variables named `'High'`, `'Low'`, `'Open'`, and `'Close'` (case insensitive).

Data Types: `double` | `table` | `timetable`

Output Arguments

`MACDLine` — MACD series

matrix | table | timetable

MACD series, returned with the same number of rows (`M`) and type (matrix, table, or timetable) as the input `Data`. The `MACDLine` is calculated by subtracting the 26-period (7.5%) exponential.

`SignalLine` — Nine-period exponential series

matrix | table | timetable

Nine-period exponential series, returned with the same number of rows (`M`) and type (matrix, table, or timetable) as the input `Data`. The nine-period (20%) exponential moving average of the `MACDLine` is used as the `SignalLine`.

More About

MACD

The MACD is calculated by subtracting the 26-period (7.5%) exponential moving average from the 12-period (15%) moving average.

The nine-period (20%) exponential moving average of the MACD line is used as the "signal" line. When the two lines are plotted, they can give you indications on when to buy or sell a stock, when overbought or oversold is occurring, and when the end of trend may occur. For example, when the MACD and the 20-day moving average line have crossed and the MACD line becomes below the other line, it is time to sell.

Version History

Introduced before R2006a

R2023a: `fints` support removed for `Data` input argument

Behavior changed in R2023a

`fints` object support for the `Data` input argument is removed.

R2022b: Support for negative price data

Behavior changed in R2022b

The `Data` input accepts negative prices.

References

[1] Achelis, S. B. *Technical Analysis from A to Z.* Second Edition. McGraw-Hill, 1995, pp. 166–168.

See Also

`adline` | `timetable` | `table` | `willad`

Topics

"Use Timetables in Finance" on page 11-7
"Convert Financial Time Series Objects (fints) to Timetables" on page 11-2

maxdrawdown

Compute maximum drawdown for one or more price series

Syntax

```
MaxDD = maxdrawdown(Data)
MaxDD = maxdrawdown( ___ ,Format)
[MaxDD,MaxDDIndex] = maxdrawdown( ___ )
```

Description

`MaxDD = maxdrawdown(Data)` 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.

`MaxDD = maxdrawdown( ___ ,Format)` adds an optional argument for `Format`.

`[MaxDD,MaxDDIndex] = maxdrawdown( ___ )` adds an optional output for `MaxDDIndex`.

Examples

Calculate Maximum Drawdown

Calculate the maximum drawdown (MaxDD) using example data with a fund, market, and cash series:

```
load FundMarketCash
MaxDD = maxdrawdown(TestData)
```

```
MaxDD = 1×3

    0.1658    0.3381         0
```

The maximum drop in the given time period was 16.58% for the fund series, and 33.81% for the market series. There was no decline in the cash series, as expected, because the cash account never loses value.

`maxdrawdown` also returns the indices (MaxDDIndex) of the maximum drawdown intervals for each series in an optional output argument.

```
[MaxDD, MaxDDIndex] = maxdrawdown(TestData)
```

```
MaxDD = 1×3

    0.1658    0.3381         0

MaxDDIndex = 2×3

     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 NaNs because it never has a drawdown.

The 16.58% value loss from month 2 to month 18 for the fund series is verified using the reported indices.

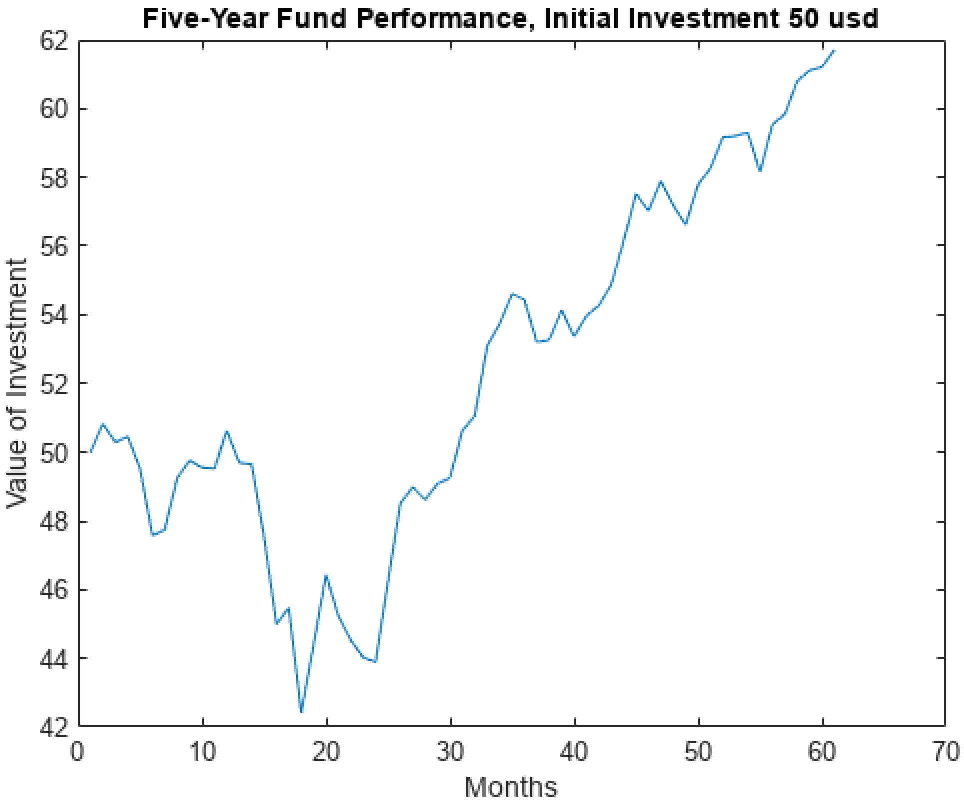
```
Start = MaxDDIndex(1,:);
End = MaxDDIndex(2,:);
(TestData(Start(1),1) - TestData(End(1),1))/TestData(Start(1),1)
```

```
ans = 0.1658
```

```
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, you can 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, 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, compute the maximum drop in value, using the `'arithmetic'` argument.

```
[MaxDD50Arith, Ind50Arith] = maxdrawdown(Fund50,'arithmetic')
```

```
MaxDD50Arith = 8.4285

Ind50Arith = 2×1

     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, from the 2nd to 18th month, is the same independently of whether drawdown is measured as return or as dollar value loss.

```
[MaxDD50LogRet, Ind50LogRet] = maxdrawdown(Fund50,'geometric')
```

```
MaxDD50LogRet = 0.1813

Ind50LogRet = 2×1

     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
```

Input Arguments

Data — Total return price series
matrix

Total return price series, specified as a `T`-by-`N` matrix with `T` samples of `N` total return price series.

Data Types: `double`

Format — (Optional) Format of Data
character vector with value of `'return'`, `'arithmetic'`, or `'geometric'`

Format of `Data`, specified as character vector with the following possible values:

- `'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

  $$dS(t) = \mu_0 S(t)dt + \sigma_0 S(t)dW(t)$$

Data Types: `char`

Output Arguments

`MaxDD` — Maximum drawdown

vector

Maximum drawdown, returned as a `1`-by-`N` vector with maximum drawdown for each of `N` time series.

Note

- 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.

`MaxDDIndex` — Start and end indexes for each maximum drawdown period for each total equity time series

vector

Start and end indexes for each maximum drawdown period for each total equity time series, returned as a `2`-by-`N` vector of start and end indexes. The first row of the vector contains the start indexes and the second row contains the end indexes of each maximum drawdown period.

Version History

Introduced in R2006b

References

[1] 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

"Performance Metrics Overview" on page 7-2

medprice

Median price from the series of high and low prices

Syntax

```
MedianPrice = medprice(Data)
```

Description

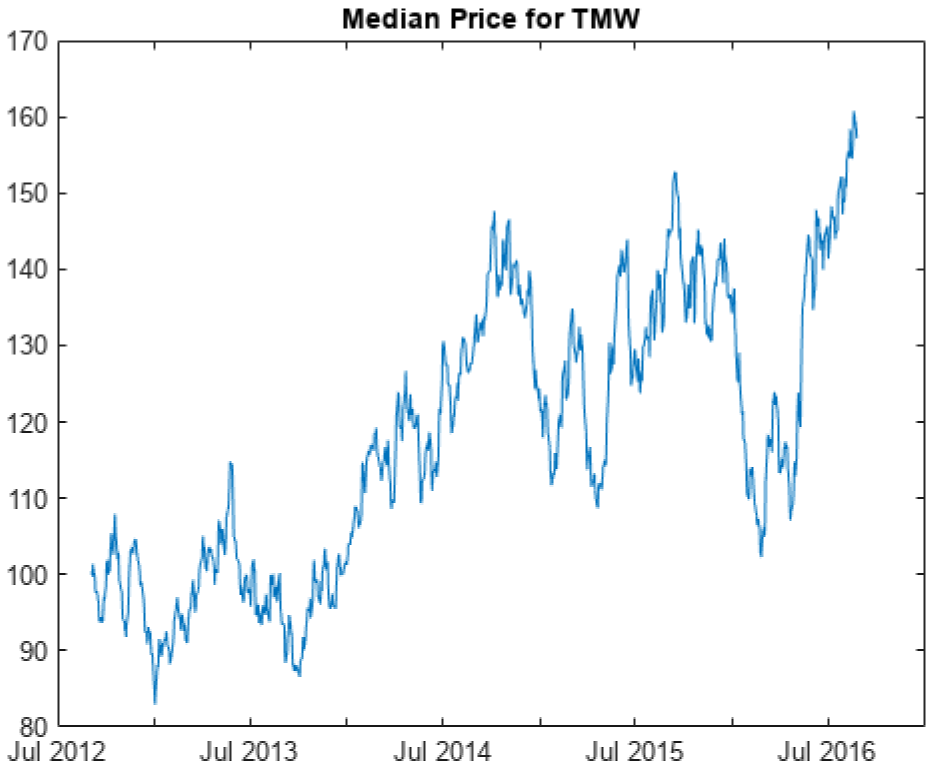
`MedianPrice = medprice(Data)` calculates the median prices from the series of high and low prices. The median price is the average of the high and low prices for each period.

Examples

Calculate the Median Prices for a Data Series for a Stock

Load the file `SimulatedStock.mat`, which provides a timetable (`TMW`) for financial data for TMW stock.

```
load SimulatedStock.mat
MedianPrice = medprice(TMW);
plot(MedianPrice.Time,MedianPrice.MedianPrice)
title('Median Price for TMW')
```



Input Arguments

`Data` — Data for high and low prices

matrix | table | timetable

Data for high and low prices, specified as a matrix, table, or timetable. For matrix input, `Data` is an `M`-by-2 matrix of high and low prices stored in the corresponding columns. The columns can be in either order because the median price is the average of the high and low prices for each period. Timetables and tables with `M` rows must contain a variable named `'High'` and `'Low'` (case insensitive).

Data Types: `double` | `table` | `timetable`

Output Arguments

`MedianPrice` — Median price series

matrix | table | timetable

Median price series, returned with the same number of rows (`M`) and the same type (matrix, table, or timetable) as the input `Data`.

Version History

Introduced before R2006a

R2023a: `fints` support removed for `Data` input argument

Behavior changed in R2023a

`fints` object support for the `Data` input argument is removed.

R2022b: Support for negative price data

Behavior changed in R2022b

The `Data` input accepts negative prices.

References

[1] Achelis, S. B. *Technical Analysis from A to Z.* Second Edition. McGraw-Hill, 1995, pp. 177–178.

See Also

`timetable` | `table` | `wclose` | `typprice`

Topics

"Use Timetables in Finance" on page 11-7
"Convert Financial Time Series Objects (fints) to Timetables" on page 11-2

mirr

Modified internal rate of return

Syntax

```
Return = mirr(CashFlow,FinRate,Reinvest)
```

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

Compute Modified Internal Rate of Return

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)
```

```
Return = 0.0832
```

Input Arguments

`CashFlow` — Cash flow
vector | matrix

Cash flow, specified as a vector or matrix. The first entry is the initial investment. If `CashFlow` is entered as a matrix, each column is treated as a separate cash flow.

Data Types: `double`

`FinRate` — Finance rate for negative cash flow values
decimal

Finance rate for negative cash flow values, specified as a decimal.

Data Types: `double`

`Reinvest` — Reinvestment rate for positive cash flow values
decimal

Reinvestment rate for positive cash flow values, specified as a decimal.

Data Types: `double`

Output Arguments

`Return` — Modified internal rate of return
numeric

Modified internal rate of return, returned as a scalar or vector.

Version History

Introduced before R2006a

References

[1] Brealey and Myers. *Principles of Corporate Finance.* McGraw-Hill Higher Education, Chapter 5, 2003.

[2] Hazen G. “A New Perspective on Multiple Internal Rates of Return.” *The Engineering Economist.* Vol. 48-1, 2003, pp. 31–51.

See Also

`annurate` | `effrr` | `irr` | `nomrr` | `pvvar` | `xirr`

Topics

“Analyzing and Computing Cash Flows” on page 2-11

movavg

Moving average of a financial time series

Note The syntax for `movavg` has changed. There is no longer support for the input arguments `Lead` and `Lag`, only a single `windowSize` is supported, and there is only one output argument (`ma`). If you want to compute the leading and lagging moving averages, you need to run `movavg` twice and adjust the `windowSize`.

Syntax

```
ma = movavg(Data,type,windowSize)
ma = movavg(___,Initialpoints)

ma = movavg(Data,type,weights)
ma = movavg(___,Initialpoints)
```

Description

`ma = movavg(Data,type,windowSize)` computes the moving average (MA) of a financial time series.

`ma = movavg(___,Initialpoints)` adds an optional argument for `Initialpoints`.

`ma = movavg(Data,type,weights)` computes the moving average (MA) of a financial time series using a `'custom'` `type` and `weights`.

`ma = movavg(___,Initialpoints)` adds an optional argument for `Initialpoints`.

Examples

Calculate the Moving Average for a Data Series

Load the file `SimulatedStock.mat`, which provides a timetable (`TMW`) for financial data.

```
load SimulatedStock.mat
type = 'linear';
windowSize = 14;
ma = movavg(TMW_CLOSE,type,windowSize)
```

```
ma = 1000×1

  100.2500
  100.3433
  100.8700
  100.4916
   99.9937
   99.3603
   98.8769
   98.6364
   98.4348
   97.8491
      ⋮
```

Calculate the Leading and Lagging Moving Averages for a Data Series

Load the file `SimulatedStock.mat`, which provides a timetable (`TMW`) for financial data.

```
load SimulatedStock.mat
type = 'linear';
malag=movavg(TMW_CLOSE,type,20) % Lagging moving average
```

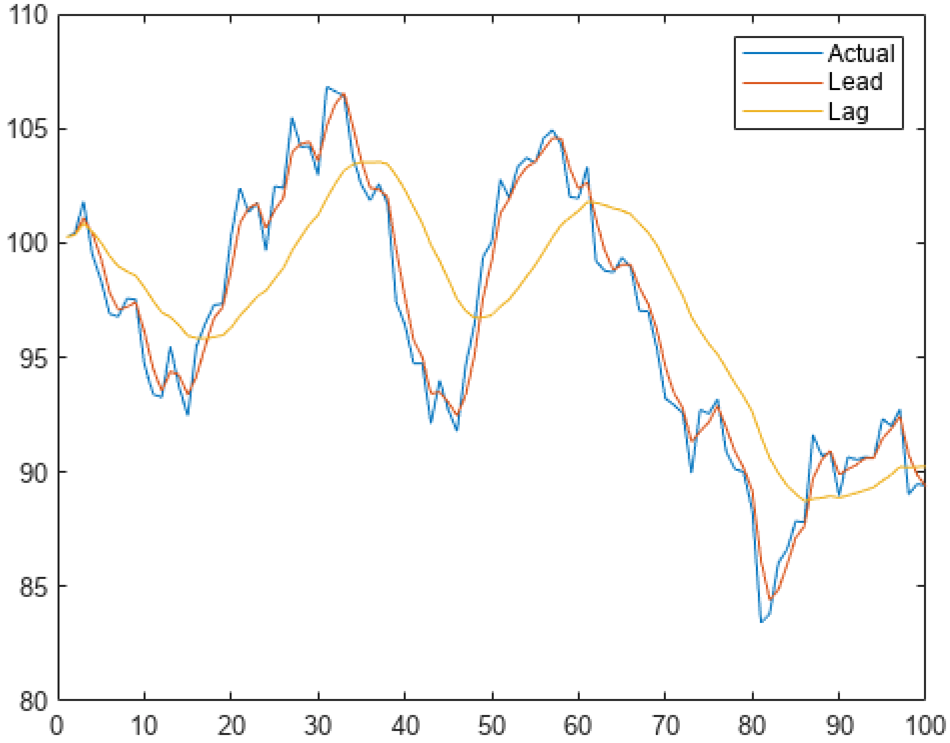
```
malag = 1000×1

  100.2500
  100.3423
  100.8574
  100.4943
  100.0198
   99.4230
   98.9728
   98.7509
   98.5688
   98.0554
      ⋮
```

```
malead=movavg(TMW_CLOSE,type,3) % Leading moving average
```

```
malead = 1000×1

  100.2500
  100.3580
  101.0900
  100.4300
   99.3183
   97.8217
   97.0833
   97.1950
   97.4133
   96.1133
      ⋮
```

Plot the leading and lagging moving averages.

```
plot(TMW_CLOSE(1:100))
hold on
plot(malead(1:100))
plot(malag(1:100))
hold off
legend('Actual','Lead','Lag')
```



Input Arguments

`Data` — Data for a financial series

matrix | table | timetable

Data for a financial series, specified as a column-oriented matrix, table, or timetable. Timetables and tables must contain variables with only a numeric type.

Data Types: `double` | `table` | `timetable`

`type` — Type of moving average to compute

character vector with value of `'simple'`, `'square-root'`, `'linear'`, `'square'`, `'exponential'`, `'triangular'`, `'modified'`, or `'custom'` | string with value of `"simple"`, `"square-root"`, `"linear"`, `"square"`, `"exponential"`, `"triangular"`, `"modified"`, or `"custom"`

Type of moving average to compute, specified as a character vector or string with an associated value.

Data Types: `char` | `string`

`windowSize` — Number of observations of the input series to include in moving average

positive integer

Number of observations of the input series to include in moving average, specified as a scalar positive integer. The observations include (`windowSize` - 1) previous data points and the current data point.

Note The windowSize argument applies only to moving averages whose type is 'simple', 'square-root', 'linear', 'square', 'exponential', 'triangular', or 'modified'.

Data Types: double

weights — Custom weights used to compute moving average
vector

Custom weights used to compute the moving average, specified as a vector.

Note The length of weights (*N*) determines the size of the moving average window (windowSize). The weights argument applies only to a 'custom' type of moving average.

To compute moving average with custom weights, the weights (w) are first normalized such that they sum to one:

```
W(i) = w(i)/sum(w), for i = 1,2,...,N
```

The normalized weights (W) are then used to form the *N*-point weighted moving average (*y*) of the input Data (*x*):

```
y(t) = W(1)*x(t) + W(2)*x(t-1) + ... + W(N)*x(t-N)
```

The initial moving average values within the window size are then adjusted according to the method specified in the name-value pair argument Initialpoints.

Data Types: double

Initialpoints — Indicates how moving average is calculated at initial points
'shrink' (default) | character vector with values of 'shrink', 'fill', or 'zero' | string with values of "shrink", "fill", or "zero"

(Optional) Indicates how the moving average is calculated at initial points (before there is enough data to fill the window), specified as a character vector or string using one of the following values:

- 'shrink' - Initializes the moving average such that the initial points include only observed data
- 'zero' - Initializes the initial points with 0
- 'fill' - Fills initial points with NaNs

Note The Initialpoints argument applies to all type specifications except for the 'exponential' and 'modified' options.

Data Types: char | string

Output Arguments

ma — Moving average series
matrix | table | timetable

Moving average series, returned with the same number of rows (`M`) and the same type (matrix, table, or timetable) as the input `Data`.

Version History

Introduced before R2006a

R2023a: `fints` support removed for `Data` input argument

Behavior changed in R2023a

`fints` object support for the `Data` input argument is removed.

References

[1] Achelis, S. B. *Technical Analysis from A to Z.* Second Edition. McGraw-Hill, 1995, pp. 184–192.

See Also

`timetable` | `table` | `bollinger` | `candle` | `dateaxis` | `highlow` | `pointfig`

Topics

"Use Timetables in Finance" on page 11-7
"Convert Financial Time Series Objects (fints) to Timetables" on page 11-2

mvnrfish

Fisher information matrix for multivariate normal or least-squares regression

Syntax

```
Fisher = mvnrfish(Data,Design,Covariance)
Fisher = mvnrfish( ___ ,MatrixFormat,CovarFormat)
```

Description

`Fisher = mvnrfish(Data,Design,Covariance)` 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.

`Fisher = mvnrfish( ___ ,MatrixFormat,CovarFormat)` computes a Fisher information matrix based on current maximum likelihood or least-squares parameter estimates using optional arguments.

Input Arguments

`Data` — Data sample

matrix

Data sample, specified as an `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 `mvnrmle` to handle missing data.)

Data Types: `double`

`Design` — Model design

matrix | cell array of character vectors

Model design, specified as 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.

Data Types: double | cell

Covariance — Estimates for the covariance of the residuals of the regression

matrix

Estimates for the covariance of the residuals of the regression, specified as an NUMSERIES-by-NUMSERIES matrix.

Data Types: double

MatrixFormat — Parameters to be included in Fisher information matrix

'full' (default) | character vector with value 'full' or 'paramonly'

(Optional) Parameters to be included in the Fisher information matrix, specified as a character vector. The choices are:

- 'full' — This is the default method that computes the full Fisher information matrix for both model and covariance parameter estimates.
- 'paramonly' — This computes only components of the Fisher information matrix associated with the model parameter estimates.

Data Types: char

CovarFormat — Format for covariance matrix

'full' (default) | character vector with value 'full' or 'diagonal'

(Optional) Format for the covariance matrix, specified as a character vector. The choices are:

- 'full' — This is the default method that computes the full covariance matrix.
- 'diagonal' — This forces the covariance matrix to be a diagonal matrix.

Data Types: char

Output Arguments

Fisher — Fisher information matrix

matrix

Fisher information matrix, returned as an TOTALPARAMS-by-TOTALPARAMS matrix.

Version History

Introduced in R2006a

See Also

mvnrstd | mvnrmle

Topics

"Multivariate Normal Regression With Missing Data" on page 9-14
"Multivariate Normal Regression Without Missing Data" on page 9-13

“Least-Squares Regression With Missing Data” on page 9-14
“Least-Squares Regression Without Missing Data” on page 9-14
“Fisher Information” on page 9-4
“Multivariate Normal Linear Regression” on page 9-2
“Least-Squares Regression” on page 9-4

mvnrmle

Multivariate normal regression (ignore missing data)

Syntax

```
[Parameters,Covariance,Resid,Info] = mvnrmle(Data,Design)
[Parameters,Covariance,Resid,Info] = mvnrmle( ___ ,MaxIterations,TolParam,
TolObj,Covar0,CovarFormat)
```

Description

`[Parameters,Covariance,Resid,Info] = mvnrmle(Data,Design)` 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, ... , `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.

`[Parameters,Covariance,Resid,Info] = mvnrmle( ___ ,MaxIterations,TolParam, TolObj,Covar0,CovarFormat)` estimates a multivariate normal regression model without missing data using optional arguments.

Input Arguments

`Data` — Data sample

matrix

Data sample, specified as an `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 `mvnrmle` to handle missing data.)

Data Types: `double`

`Design` — Model design

matrix | cell array of character vectors

Model design, specified as 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.

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.

Data Types: double | cell

MaxIterations — Maximum number of iterations for the estimation algorithm
100 (default) | numeric

(Optional) Maximum number of iterations for the estimation algorithm, specified as a numeric. The default value is 100.

Data Types: double

TolParam — Convergence tolerance for estimation algorithm
sqrt(eps) (default) | numeric

(Optional) Convergence tolerance for estimation algorithm based on changes in model parameter estimates, specified as a numeric. The 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.

Data Types: double | table | timetable

TolObj — Convergence tolerance for estimation algorithm based on changes in objective function
eps ∧ 3/4 (default) | numeric

(Optional) Convergence tolerance for estimation algorithm based on changes in the objective function, specified as a numeric. The 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.

Data Types: double

Covar0 — User-supplied initial or known estimate for the covariance matrix of the regression residuals
identity matrix (default) | matrix

(Optional) user-supplied initial or known estimate for the covariance matrix of the regression residuals, specified as an `NUMSERIES`-by-`NUMSERIES` matrix.

Data Types: `double`

`CovarFormat` — Format for covariance matrix
`'full'` (default) | character vector with value `'full'` or `'diagonal'`

(Optional) Format for the covariance matrix, specified as a character vector. The choices are:

- `'full'` — This is the default method that computes the full covariance matrix.
- `'diagonal'` — This forces the covariance matrix to be a diagonal matrix.

Data Types: `char`

Output Arguments

`Parameters` — Parameters of the regression model
vector

Parameters of the regression model, returned as an `NUMPARAMS`-by-`1` column vector of estimates for the parameters of the regression model.

`Covariance` — Covariance of the regression model's residuals
matrix

Covariance of the regression model's residuals, returned as an `NUMSERIES`-by-`NUMSERIES` matrix of estimates for the covariance of the regression model's residuals.

`Resid` — Residuals from the regression
matrix

Residuals from the regression, returned as an `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.

`Info` — Structure containing additional information from the regression
structure

Structure containing additional information from the regression, returned as a structure. 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.

Use the estimates in the output structure `Info` for diagnostic purposes.

Version History

Introduced in R2006a

References

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

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

See Also

`ecmmvnrmle` | `mvnrstd` | `mvnrobj` | `mvregress`

Topics

"Multivariate Normal Regression" on page 9-13
"Least-Squares Regression" on page 9-14
"Covariance-Weighted Least Squares" on page 9-14
"Feasible Generalized Least Squares" on page 9-15
"Seemingly Unrelated Regression" on page 9-16
"Multivariate Normal Regression With Missing Data" on page 9-14
"Multivariate Normal Regression Without Missing Data" on page 9-13
"Capital Asset Pricing Model with Missing Data" on page 9-33
"Multivariate Normal Linear Regression" on page 9-2

mvnrobj

Log-likelihood function for multivariate normal regression without missing data

Syntax

```
Objective = mvnrobj(Data,Design,Parameters,Covariance)
Objective = mvnrobj( ___ ,CovarFormat)
```

Description

`Objective = mvnrobj(Data,Design,Parameters,Covariance)` computes the log-likelihood function based on current maximum likelihood parameter estimates without missing data. `Objective` is a scalar that contains the log-likelihood function.

`Objective = mvnrobj( ___ ,CovarFormat)` computes the log-likelihood function based on current maximum likelihood parameter estimates without missing data using an optional argument.

Input Arguments

`Data` — Data sample

matrix

Data sample, specified as an `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.)

Data Types: `double`

`Design` — Model design

matrix | cell array of character vectors

Model design, specified as 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.

Data Types: `double` | `cell`

`Parameters` — Estimates for the parameters of regression model

vector

Estimates for the parameters of regression model, specified as an `NUMPARAMS`-by-`1` column vector.

Data Types: `double`

`Covariance` — Estimates for the covariance of the residuals of the regression
matrix

Estimates for the covariance of the residuals of the regression, specified as an `NUMSERIES`-by-`NUMSERIES` matrix.

Data Types: `double`

`CovarFormat` — Format for covariance matrix
`'full'` (default) | character vector with value `'full'` or `'diagonal'`

(Optional) Format for the covariance matrix, specified as a character vector. The choices are:

- `'full'` — This is the default method that computes the full covariance matrix.
- `'diagonal'` — This forces the covariance matrix to be a diagonal matrix.

Data Types: `char`

Output Arguments

`Objective` — Log-likelihood function
scalar

Log-likelihood function, returned as scalar.

Version History

Introduced in R2006a

See Also

`ecmmvnrmle` | `ecmmvnrobj` | `mvnrmle`

Topics
“Multivariate Normal Regression Without Missing Data” on page 9-13
“Multivariate Normal Regression” on page 9-13
“Least-Squares Regression” on page 9-14
“Covariance-Weighted Least Squares” on page 9-14
“Feasible Generalized Least Squares” on page 9-15
“Seemingly Unrelated Regression” on page 9-16
“Multivariate Normal Linear Regression” on page 9-2

mvnrstd

Evaluate standard errors for multivariate normal regression model

Syntax

```
[StdParameters,StdCovariance] = mvnrstd(Data,Design,Covariance)
[StdParameters,StdCovariance] = mvnrstd( ___ ,CovarFormat)
```

Description

`[StdParameters,StdCovariance] = mvnrstd(Data,Design,Covariance)` 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 $k = 1, \ldots,$ `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`.

`[StdParameters,StdCovariance] = mvnrstd( ___ ,CovarFormat)` computes the log-likelihood function based on current maximum likelihood parameter estimates without missing data using an optional argument.

Input Arguments

`Data` — Data sample

matrix

Data sample, specified as an `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.)

Data Types: `double`

`Design` — Model design

matrix | cell array of character vectors

Model design, specified as 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.

Data Types: double | cell

Covariance — Estimates for the covariance of the residuals of the regression
matrix

Estimates for the covariance of the residuals of the regression, specified as an NUMSERIES-by-NUMSERIES matrix.

Data Types: double

CovarFormat — Format for covariance matrix
'full' (default) | character vector with value 'full' or 'diagonal'

(Optional) Format for the covariance matrix, specified as a character vector. The choices are:

- 'full' — This is the default method that computes the full covariance matrix.
- 'diagonal' — This forces the covariance matrix to be a diagonal matrix.

Data Types: char

Output Arguments

StdParameters — standard errors for each element of Parameters, the vector of estimated model parameters
vector

Standard errors for each element of Parameters, the vector of estimated model parameters, returned as an NUMPARAMS-by-1 column vector.

StdCovariance — standard errors for each element of Covariance, the matrix of estimated covariance parameters
matrix

Standard errors for each element of Covariance, the matrix of estimated covariance parameters, returned as an NUMSERIES-by-NUMSERIES matrix.

Version History

Introduced in R2006a

References

[1] 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 9-13
“Multivariate Normal Regression” on page 9-13
“Least-Squares Regression” on page 9-14
“Covariance-Weighted Least Squares” on page 9-14
“Feasible Generalized Least Squares” on page 9-15
“Seemingly Unrelated Regression” on page 9-16
“Multivariate Normal Linear Regression” on page 9-2

negvolidx

Negative volume index

Syntax

```
volume = negvolidx(Data)
volume = negvolidx(___,Name,Value)
```

Description

`volume = negvolidx(Data)` calculates the negative volume index from the series of closing stock prices and trade volume.

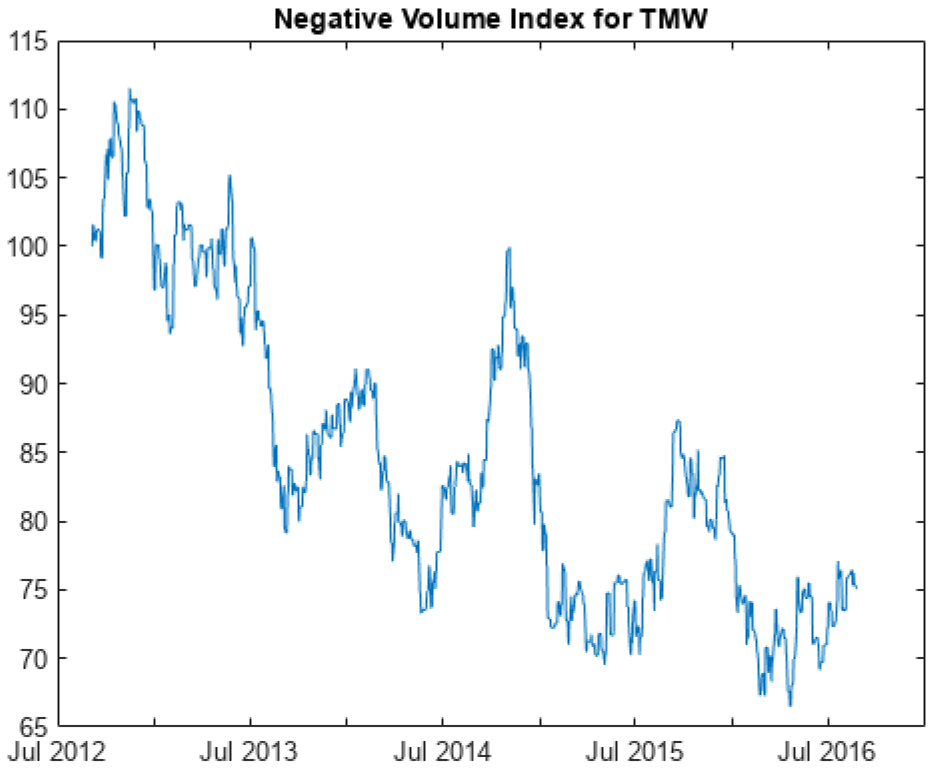
`volume = negvolidx(___,Name,Value)` adds optional name-value pair arguments.

Examples

Calculate the Negative Volume Index for a Data Series for a Stock

Load the file `SimulatedStock.mat`, which provides a timetable (`TMW`) for financial data for TMW stock.

```
load SimulatedStock.mat
volume = negvolidx(TMW);
plot(volume.Time,volume.NegativeVolume)
title('Negative Volume Index for TMW')
```



Input Arguments

`Data` — Data with closing prices and trade volume

matrix | table | timetable

Data with closing prices and trade volume, specified as a matrix, table, or timetable. For matrix input, `Data` is an `M`-by-2 with closing prices and trade volume stored in the first and second columns. Timetables and tables with `M` rows must contain variables named `'Close'` and `'Volume'` (case insensitive).

Data Types: `double` | `table` | `timetable`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `volume = negvolidx(TMW,'InitialValue',500)`

`InitialValue` — Initial value for negative volume index

`100` (default) | positive integer

Initial value for negative volume index, specified as the comma-separated pair consisting of `'InitialValue'` and a scalar positive integer.

Data Types: `double`

Output Arguments

`volume` — Negative volume index
matrix | table | timetable

Negative volume index, returned with the same number of rows (`M`) and the same type (matrix, table, or timetable) as the input `Data`.

More About

Negative Volume Index

Negative volume index shows the days when the trading volume of a particular security is substantially lower than other days.

Version History

Introduced before R2006a

R2023a: `fints` support removed for `Data` input argument
Behavior changed in R2023a

`fints` object support for the `Data` input argument is removed.

R2022b: Support for negative price data
Behavior changed in R2022b

The `Data` input accepts negative prices.

References

[1] Achelis, S. B. *Technical Analysis from A to Z.* Second Edition. McGraw-Hill, 1995, pp. 193–194.

See Also

`timetable` | `table` | `onbalvol` | `posvolidx`

Topics
“Use Timetables in Finance” on page 11-7
“Convert Financial Time Series Objects (fints) to Timetables” on page 11-2

nomrr

Nominal rate of return

Syntax

`Return = nomrr(Rate,NumPeriods)`

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
```

Input Arguments

`Rate` — Effective annual percentage rate
decimal

Effective annual percentage rate, specified as a decimal.

Data Types: `double`

`NumPeriods` — Number of compounding periods per year
scalar numeric

Number of compounding periods per year, specified as a scalar numeric.

Data Types: `double`

Output Arguments

`Return` — Nominal rate of return
numeric

Nominal rate of return, returned as a numeric.

Version History

Introduced before R2006a

See Also

`effrr | irr | mirr | taxedrr | xirr`

Topics

"Analyzing and Computing Cash Flows" on page 2-11

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 dates 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 dates 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(1899,1,1),datetime(1899,6,30))
```

```
Closures = 7x1 datetime
   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`:

```
nyseclosures(datetime(1952,9,1),datetime(1952,10,31),'a')
```

```
ans = 10x1 datetime
   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.

Input Arguments

`StartDate` — Start date

start of default date range, January 1, 1885 (default) | datetime array | string array | date character vector

Start date, specified as a scalar or a vector using a datetime array, string array, or date character vectors.

To support existing code, `nyseclosures` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

EndDate — End date

end of default date range, December 31, 2070 (default) | datetime array | string array | date character vector

End date, specified as a scalar or a vector using a datetime array, string array, or date character vectors.

To support existing code, `nyseclosures` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `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 strings or date character vectors, both `Closures` and `SatTransition` are returned as serial date numbers. Use the function `datestr` to convert serial date numbers to formatted date character vectors. 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 strings or date character vectors, both `Closures` and `SatTransition` are returned as serial date numbers. Use the function `datestr` to convert serial date numbers to formatted date character vectors. If either `StartDate` or `EndDate` are datetime arrays, both `Closures` and `SatTransition` are returned as datetime arrays.

More About

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'.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although nyseclosures supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

busdate | createholidays | fbusdate | isbusday | lbusdate | holidays | datetime

Topics

“Handle and Convert Dates” on page 2-2
“Trading Calendars User Interface” on page 12-2
“UICalendar User Interface” on page 12-4

onbalvol

On-Balance Volume (OBV)

Syntax

```
volume = onbalvol(Data)
```

Description

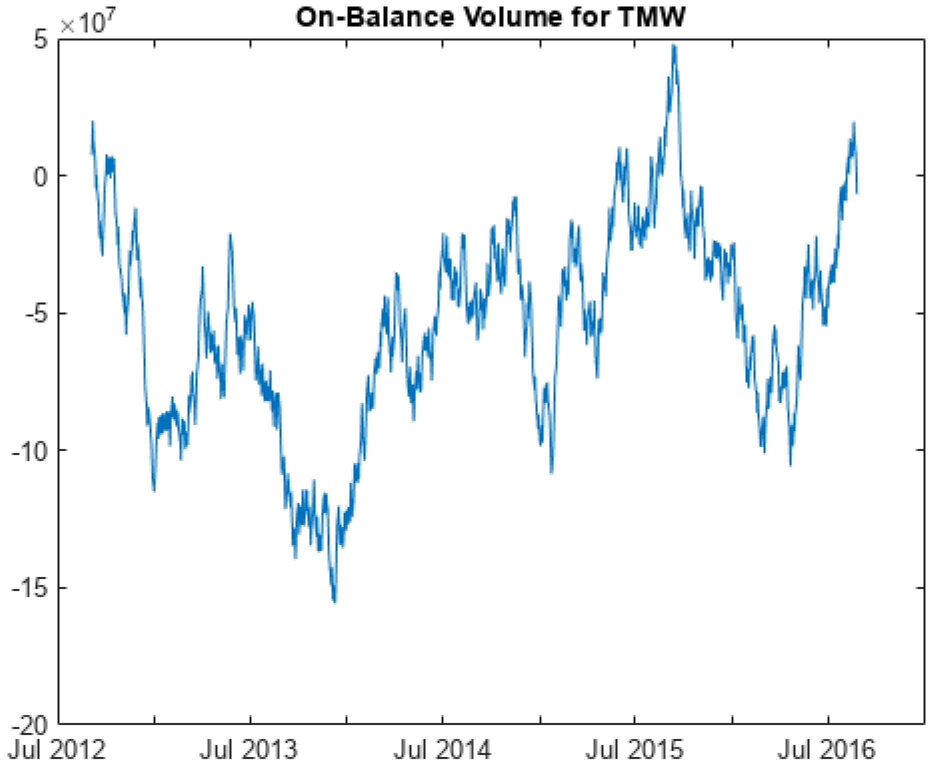
`volume = onbalvol(Data)` calculates the On-Balance Volume from the series of closing stock prices and trade volume.

Examples

Calculate the On-Balance Volume for a Data Series for a Stock

Load the file `SimulatedStock.mat`, which provides a timetable (`TMW`) for financial data for TMW stock.

```
load SimulatedStock.mat
volume = onbalvol(TMW);
plot(volume.Time,volume.OnBalanceVolume)
title('On-Balance Volume for TMW')
```



Input Arguments

`Data` — Data for closing prices and trade volume

matrix | table | timetable

Data for closing prices and trade volume, specified as a matrix, table, or timetable. For matrix input, `Data` is an `M`-by-2 matrix of closing prices and trade volume stored in the first and second columns. Timetables and tables with `M` rows must contain variables named `'Close'` and `'Volume'` (case insensitive).

Data Types: `double` | `table` | `timetable`

Output Arguments

`volume` — On-Balance Volume

matrix | table | timetable

On-Balance Volume, returned with the same number of rows (`M`) and the same type (matrix, table, or timetable) as the input `Data`.

Version History

Introduced before R2006a

R2023a: `fints` support removed for `Data` input argument

Behavior changed in R2023a

`fints` object support for the `Data` input argument is removed.

R2022b: Support for negative price data

Behavior changed in R2022b

The `Data` input accepts negative prices.

References

[1] Achelis, S. B. *Technical Analysis from A to Z.* Second Edition. McGraw-Hill, 1995, pp. 207–209.

See Also

`timetable` | `table` | `negvolidx` | `posvolidx`

Topics

"Use Timetables in Finance" on page 11-7
"Convert Financial Time Series Objects (fints) to Timetables" on page 11-2

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

Calculate the Profit of an Option

This example shows how to return the profit of an option. For example, consider buying (going long on) a call option with a strike price of $90 on an underlying asset with a current price of $100 for a cost of $4.

```
Profit = opprofit(100, 90, 4, 0, 0)
```

```
Profit = 6
```

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.

Version History

Introduced before R2006a

See Also

binprice | blsprice

Topics

“Pricing and Analyzing Equity Derivatives” on page 2-39
“Greek-Neutral Portfolios of European Stock Options” on page 10-14
“Plotting Sensitivities of an Option” on page 10-25
“Plotting Sensitivities of a Portfolio of Options” on page 10-27

payadv

Periodic payment given number of advance payments

Syntax

```
Payment = payadv(Rate,NumPeriods,PresentValue,FutureValue,Advance)
```

Description

`Payment = payadv(Rate,NumPeriods,PresentValue,FutureValue,Advance)` computes 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
```

Input Arguments

`Rate` — Lending or borrowing rate per period
decimal

Lending or borrowing rate per period, specified as a decimal. The `Rate` must be greater than or equal to `0`.

Data Types: `double`

`NumPeriods` — Number of periods in the life of the instrument
integer

Number of periods in the life of the instrument, specified as an integer.

Data Types: `double`

`PresentValue` — Present value of instrument
numeric

Present value of the instrument, specified as a numeric.

Data Types: `double`

`FutureValue` — Future value or target value attained after `NumPeriods` periods
numeric

Future value or target value to be attained after `NumPeriods` periods, specified as a numeric.

Data Types: `double`

`Advance` — Number of advance payments

integer

Number of advance payments, specified as an integer. If the payments are made at the beginning of the period, add `1` to `Advance`.

Data Types: `double`

Output Arguments

`Payment` — Periodic payment

numeric

Periodic payment, returned as the periodic payment given a number of advance payments.

Version History

Introduced before R2006a

See Also

`amortize` | `payodd` | `payper`

Topics

"Analyzing and Computing Cash Flows" on page 2-11

payodd

Payment of loan or annuity with odd first period

Syntax

```
Payment = payodd(Rate,NumPeriods,PresentValue,FutureValue,Days)
```

Description

`Payment = payodd(Rate,NumPeriods,PresentValue,FutureValue,Days)` computes 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
```

Input Arguments

`Rate` — Interest rate per period
decimal

Interest rate per period, specified as a decimal.

Data Types: `double`

`NumPeriods` — Number of periods in the life of the instrument
integer

Number of periods in the life of the instrument, specified as an integer.

Data Types: `double`

`PresentValue` — Present value of instrument
numeric

Present value of the instrument, specified as a numeric.

Data Types: `double`

`FutureValue` — Future value or target value attained after `NumPeriods` periods
numeric

Future value or target value to be attained after `NumPeriods` periods, specified as a numeric.

Data Types: `double`

`Days` — Actual number of days until first payment is made
integer

Actual number of days until the first payment is made, specified as an integer.

Data Types: `double`

Output Arguments

`Payment` — Payment
numeric

Payment, returns the payment for a loan or annuity with an odd first period.

Version History

Introduced before R2006a

See Also

`amortize` | `payadv` | `payper`

Topics

"Analyzing and Computing Cash Flows" on page 2-11

payper

Periodic payment of loan or annuity

Syntax

```
Payment = payper(Rate,NumPeriods,PresentValue)
Payment = payper( ___ ,FutureValue,Due)
```

Description

`Payment = payper(Rate,NumPeriods,PresentValue)` returns the periodic payment of a loan or annuity.

`Payment = payper( ___ ,FutureValue,Due)` adds optional arguments.

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
```

Input Arguments

`Rate` — Interest rate per period

decimal

Interest rate per period, specified as a decimal.

Data Types: `double`

`NumPeriods` — Number of payment periods in life of instrument

integer

Number of payment periods in the life of instrument, specified as an integer.

Data Types: `double`

`PresentValue` — Present value of the instrument

numeric

Present value of the instrument, specified as a numeric.

Data Types: `double`

`FutureValue` — Future value or target value attained after `NumPeriods` periods

`0` (default) | numeric

(Optional) Future value or target value to be attained after `NumPeriods` periods, specified as a numeric.

Data Types: `double`

`Due` — Indicator for when payments are due

`0` (default) | logical with value of `1` or `0`

(Optional) Indicator for when payments are due, specified as a logical with a value of `0` = end of period (default), or `1` = beginning of period.

Data Types: `logical`

Output Arguments

`Payment` — Periodic payment

numeric

Periodic payment, returns the periodic payment of a loan or annuity.

More About

Annuity

An annuity is 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.

Version History

Introduced before R2006a

See Also

`fvfix` | `amortize` | `payadv` | `payodd` | `pvfix`

Topics

"Analyzing and Computing Cash Flows" on page 2-11

payuni

Uniform payment equal to varying cash flow

Syntax

```
Series = payuni(CashFlow,Rate)
```

Description

`Series = payuni(CashFlow,Rate)` computes the uniform series value of a varying cash flow.

Examples

Calculate the Uniform Series Value

This example shows how to calculate the uniform series value using `payuni`.

The following 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)
```

```
Series = 429.6296
```

Input Arguments

`CashFlow` — Cash flows
vector

Cash flows, specified as a vector of varying cash flows. Include the initial investment as the initial cash flow value (a negative number).

Data Types: `double`

`Rate` — Periodic interest rate
decimal

Periodic interest rate, specified as a decimal.

Data Types: `double`

Output Arguments

`Series` — Uniform series

numeric

Uniform series, returned as the value of a varying cash flow.

Version History

Introduced before R2006a

See Also

`fvfix` | `pvfix` | `fvvar` | `irr` | `pvvar`

Topics

“Analyzing and Computing Cash Flows” on page 2-11

pcalims

Linear inequalities for individual asset allocation

Syntax

```
[A,b] = pcalims(AssetMin,AssetMax)
[A,b] = pcalims( ___ ,NumAssets)
```

Description

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 4-17.

`[A,b] = pcalims(AssetMin,AssetMax)` specifies the lower and upper bounds of portfolio allocations in each of `NumAssets` available asset investments. `pcalims` specifies the lower and upper bounds of portfolio allocations in each of `NASSETS` available asset investments.

Note If `pcalims` is called with fewer than two output arguments, the function returns `A` concatenated with `b` `[A,b]`.

`[A,b] = pcalims( ___ ,NumAssets)` specifies options using an optional argument in addition to the input arguments in the previous syntax.

Examples

Compute Linear Inequalities for Individual Asset Allocation

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.

Minimum weight:

- IBM — 0
- INTC — 0
- CSCO — 0

Maximum weight:

- IBM — 0.5
- INTC —
- CSCO — 0.8

```
AssetMin = 0
AssetMin = 0
```

```
AssetMax = [0.5 NaN 0.8]
```

```
AssetMax = 1×3

    0.5000       NaN    0.8000
```

```
[A,b] = pcalims(AssetMin, AssetMax)
```

```
A = 5×3

     1     0     0
     0     0     1
    -1     0     0
     0    -1     0
     0     0    -1

b = 5×1

    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`.

Minimum weight:

- IBM — 0
- INTC — 0
- CSCO — 0

Maximum weight:

- IBM — 1
- INTC — 1
- CSCO — 1

```
AssetMin = 0
```

```
AssetMin = 0
```

```
AssetMax = 1
```

```
AssetMax = 1
```

```
NumAssets = 3
```

```
NumAssets = 3
```

```
[A,b] = pcalims(AssetMin, AssetMax, NumAssets)
```

```
A = 6×3

     1     0     0
     0     1     0
     0     0     1
    -1     0     0
     0    -1     0
     0     0    -1


b = 6×1

     1
     1
     1
     0
     0
     0
```

Portfolio weights of 50% in IBM and 50% in INTC satisfy the constraints.

Input Arguments

`AssetMin` — Minimum allocations in each asset

scalar numeric | vector

Minimum allocations in each asset, specified as a scalar numeric or `NASSETS` vector. `NaN` indicates no constraint.

Data Types: `double`

`AssetMax` — Maximum allocations in each asset

scalar numeric | vector

Maximum allocations in each asset, specified as a scalar numeric or `NASSETS` vector. `NaN` indicates no constraint.

Data Types: `double`

`NumAssets` — Number of assets

length of `AssetMin` or `AssetMax` (default) | scalar numeric

(Optional) Number of assets, specified as a scalar numeric.

Data Types: `double`

Output Arguments

A — Lower bound

matrix

Lower bound, returned as a matrix such that `A*PortWts' <= b`, where `PortWts` is a 1-by-`NASSETS` vector of asset allocations.

b — Upper bound
vector

Upper bound, returned as a vector such that `A*PortWts' <= b`, where `PortWts` is a 1-by-`NASSETS` vector of asset allocations.

Version History

Introduced before R2006a

See Also

`pcgcomp` | `pcglims` | `portstats` | `pcpval` | `portcons` | `portopt` | `Portfolio`

Topics

“Portfolio Construction Examples” on page 3-5
“Portfolio Selection and Risk Aversion” on page 3-7
“Active Returns and Tracking Error Efficient Frontier” on page 3-25
“Analyzing Portfolios” on page 3-2
“Portfolio Optimization Functions” on page 3-3

pcgcomp

Linear inequalities for asset group comparison constraints

Syntax

```
[A,b] = pcgcomp(GroupA,AtoBmin,AtoBmax,GroupB)
```

Description

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 4-17.

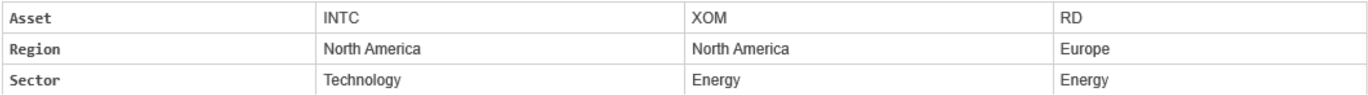
`[A,b] = pcgcomp(GroupA,AtoBmin,AtoBmax,GroupB)` specifies that the ratio of allocations in one group to allocations in another group is at least `AtoBmin` to `1` and at most `AtoBmax` to `1`. Comparisons can be made between an arbitrary number of group pairs `NGROUPS` comprising subsets of `NASSETS` available investments.

If `pcgcomp` is called with fewer than two output arguments, the function returns `A` concatenated with `b` `[A,b]`.

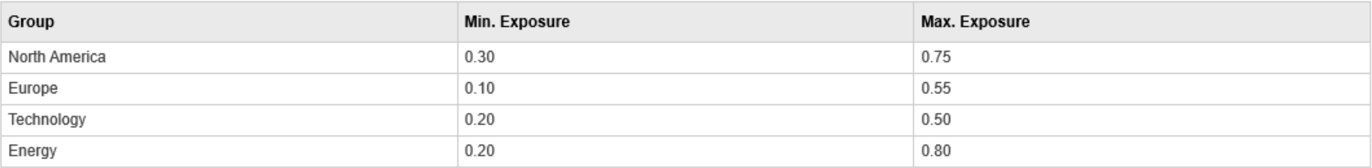
Examples

Linear Inequalities for Asset Group-to-Group Comparison Constraints

Use the following assets and groupings.



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 = 2×3

    0.2000   -0.8000         0
   -0.2000    0.8000         0


b = 2×1

     0
     0
```

Portfolio weights of 40% for INTC, 10% for XOM, and 50% for RD satisfy the constraints.

Input Arguments

`GroupA` — Grouping A

vector

Grouping A, specified as a number of groups (`NGROUPS`) by number of assets (`NASSETS`) vector 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`.

Data Types: `double`

`AtoBmin` — Minimum ratios

scalar | vector

Minimum ratios, specified as a scalar or `NGROUPS`-long vectors of minimum 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`.

Data Types: `double`

`AtoBmax` — Maximum ratios

scalar | vector

Maximum ratios, specified as a scalar or `NGROUPS`-long vectors of 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`.

Data Types: `double`

`GroupB` — Grouping A

vector

Grouping B, specified as a number of groups (`NGROUPS`) by number of assets (`NASSETS`) vector 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`.

Data Types: `double`

Output Arguments

A — Lower bound

matrix

Lower bound, returned as a matrix such that `A*PortWts' <= b`, where `PortWts` is a 1-by-`NASSETS` vector of asset allocations.

b — Upper bound

vector

Upper bound, returned as a vector such that `A*PortWts' <= b`, where `PortWts` is a 1-by-`NASSETS` vector of asset allocations.

Version History

Introduced before R2006a

See Also

`pcalims` | `pcglims` | `pcpval` | `portcons` | `portopt` | `Portfolio`

Topics

"Portfolio Construction Examples" on page 3-5
"Portfolio Selection and Risk Aversion" on page 3-7
"Active Returns and Tracking Error Efficient Frontier" on page 3-25
"Analyzing Portfolios" on page 3-2
"Portfolio Optimization Functions" on page 3-3

pcglims

Linear inequalities for asset group minimum and maximum allocation

Syntax

```
[A,b] = pcglims(Groups,GroupMin,GroupMax)
```

Description

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 4-17.

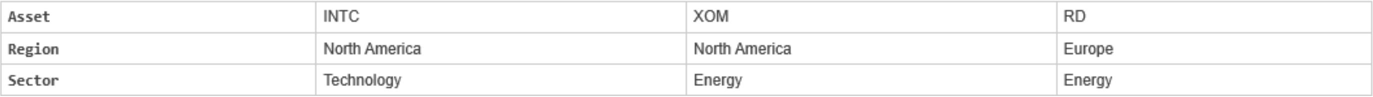
`[A,b] = pcglims(Groups,GroupMin,GroupMax)` specifies minimum and maximum allocations to groups of assets. Bounds can be specified for an arbitrary number of groups `NGROUPS`,made up as subsets of the `NASSETS` investments.

If `pcglims` is called with fewer than two output arguments, the function returns `A` concatenated with `b [A,b]`.

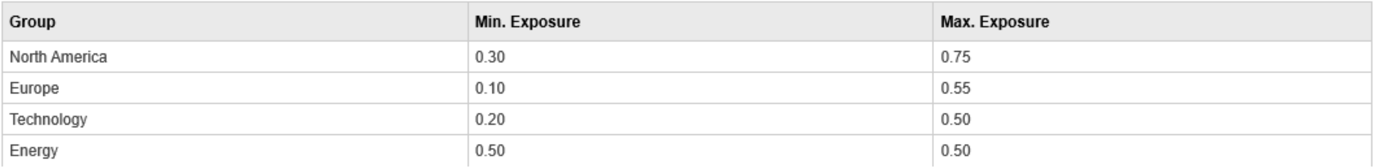
Examples

Linear Inequalities for Asset Group Minimum and Maximum Allocation

Use the following assets and groupings.



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 = 8×3

    -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 = 8×1

   -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.

Input Arguments

Groups — Grouping

vector

Grouping, specified as a number of groups (NGROUPS) by number of assets (NASSETS) vector 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.

Data Types: double

GroupMin — Minimum allocations

scalar | vector

Minimum allocations, specified as a scalar or NGROUPS-long vectors of minimum combined allocations in each group. NaN indicates no constraint. Scalar bounds are applied to all groups.

Data Types: double

GroupMax — Maximum allocations

scalar | vector

Maximum allocations, specified as a scalar or NGROUPS-long vectors maximum combined allocations in each group. NaN indicates no constraint. Scalar bounds are applied to all groups.

Data Types: double

Output Arguments

A — Lower bound

matrix

Lower bound, returned as a matrix such that A*PortWts' <= b, where PortWts is a 1-by-NASSETS vector of asset allocations.

b — Upper bound

vector

Upper bound, returned as a vector such that A*PortWts' <= b, where PortWts is a 1-by-NASSETS vector of asset allocations.

Version History

Introduced before R2006a

See Also

pcalims | pcgcomp | pcpval | portcons | portopt | Portfolio

Topics

“Portfolio Construction Examples” on page 3-5
“Portfolio Selection and Risk Aversion” on page 3-7
“Active Returns and Tracking Error Efficient Frontier” on page 3-25
“Analyzing Portfolios” on page 3-2
“Portfolio Optimization Functions” on page 3-3

pcpval

Linear inequalities for fixing total portfolio value

Syntax

```
[A,b] = pcpval(PortValue,NumAssets)
```

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`.

Note As an alternative to `pcpval`, use the `Portfolio` object (`Portfolio`) for mean-variance portfolio optimization. The `Portfolio` 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 4-17.

Examples

Scale the Value of a Portfolio

Scale the value of a portfolio of three assets that are equal to 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 = 2×3

     1     1     1
    -1    -1    -1


b = 2×1

     1
    -1
```

Portfolio weights of 40%, 10%, and 50% in the three assets satisfy the constraints.

Input Arguments

PortValue — Total value of asset portfolio

numeric

Total value of asset portfolio, specified as a scalar numeric representing the 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.

Data Types: `double`

NumAssets — Number of available asset investments

numeric

Number of available asset investments, specified as a scalar numeric.

Data Types: `double`

Output Arguments

A — Asset allocations

matrix

Asset allocations, returned as a matrix such that `A*PortWts' <= b`, where `PortWts` is a `1`-by-`NASSETS` vector of asset allocations.

b — Asset allocations

vector

Asset allocations, returned as a vector such that `A*PortWts' <= b`, where `PortWts` is a `1`-by-`NASSETS` vector of asset allocations.

Note If `pcpcval` is called with fewer than two output arguments, returns `A` and `b` are concatenated together:

```
Cons = [A, b];
Cons = pcpval(PortValue, NumAssets)
```

Version History

Introduced before R2006a

See Also

`pcalims` | `pcgcomp` | `pcglims` | `portcons` | `portopt` | `Portfolio`

Topics

"Portfolio Construction Examples" on page 3-5
"Portfolio Selection and Risk Aversion" on page 3-7
"Active Returns and Tracking Error Efficient Frontier" on page 3-25
"Analyzing Portfolios" on page 3-2
"Portfolio Optimization Functions" on page 3-3

periodicreturns

Periodic total returns from total return prices

Syntax

```
TotalReturn = periodicreturns(TotalReturnPrices)
TotalReturn = periodicreturns(___,Period)
```

Description

`TotalReturn = periodicreturns(TotalReturnPrices)` calculates the daily total returns from a daily total return price series.

`TotalReturn = periodicreturns(___,Period)` specifies options using one or more optional arguments in addition to the input arguments in the previous syntax.

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=9×2 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
```

Input Arguments

`TotalReturnPrices` — Total return prices for a given security
matrix | table

Total return prices for a given security, specified as an `NUMOBS`-by-`NASSETS + 1` matrix where Column 1 contains MATLAB serial date numbers. The remaining columns contain total return price data.

If you specify `TotalReturnPrices` as a table, the first column of the table represents the dates (as either a datetime array, string array, date character vectors, or serial date numbers) while the other columns represent the returns data. If a table is used, `TotalReturn` is returned as a table.

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 of `TotalReturn`, and the most recent date in the last row of `TotalReturn`.

Data Types: `double` | `datetime` | `string` | `char` | `table`

`Period` — Periodicity flag used to compute total returns
`'d'` (default) | character vector | numeric

(Optional) Periodicity flag used to compute total returns, specified as one of the following values:

- `'d'` = daily values (default)
- `'w'` = weekly values
- `'m'` = monthly values
- *n* = rolling return periodic values, where *n* is an integer

Data Types: `char` | `double`

Output Arguments

`TotalReturn` — Total return values
matrix | table

Total return values, returned as a `P`-by-`N` matrix or table consisting of either Dates in column 1 and daily return values in the remaining columns or period-end dates in column 1 and monthly return values in the remaining columns. The format of `TotalReturn` matches the format of the input `TotalReturnPrices`.

Version History

Introduced before R2006a

See Also

`totalreturnprice`

Topics
“Portfolio Construction Examples” on page 3-5
“Portfolio Optimization Functions” on page 3-3

plotFrontier

Plot efficient frontier

Syntax

```
[prsk,pret] = plotFrontier(obj)

[prsk,pret] = plotFrontier(obj,NumPorts)
[prsk,pret] = plotFrontier(obj,PortWeights)
[prsk,pret] = plotFrontier(obj,PortRisk,PortReturn)
```

Description

`[prsk,pret] = plotFrontier(obj)` estimates the efficient frontier with default number of 10 portfolios on the frontier, and plots the corresponding efficient frontier for `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` objects. For details on the respective workflows when using these different objects, see “Portfolio Object Workflow” on page 4-17, “PortfolioCVaR Object Workflow” on page 5-16, and “PortfolioMAD Object Workflow” on page 6-15.

`[prsk,pret] = plotFrontier(obj,NumPorts)` estimates the efficient frontier with a specified number portfolios on the frontier, and plots the corresponding efficient frontier. The number of portfolios is defined by `NumPorts`.

`[prsk,pret] = plotFrontier(obj,PortWeights)` estimates efficient portfolio risks and returns with `PortWeights`, and plots the efficient frontier with those portfolios. This syntax assumes that you provide valid efficient portfolio weights as input. `PortWeights` is a `NumAsset`-by-`NumPorts` matrix.

`[prsk,pret] = plotFrontier(obj,PortRisk,PortReturn)` plots the efficient frontier with the given risks and returns. This syntax assumes that you provide valid inputs for efficient portfolio risks and returns. `PortRisk` and `PortReturn` are vectors with the same size.

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.

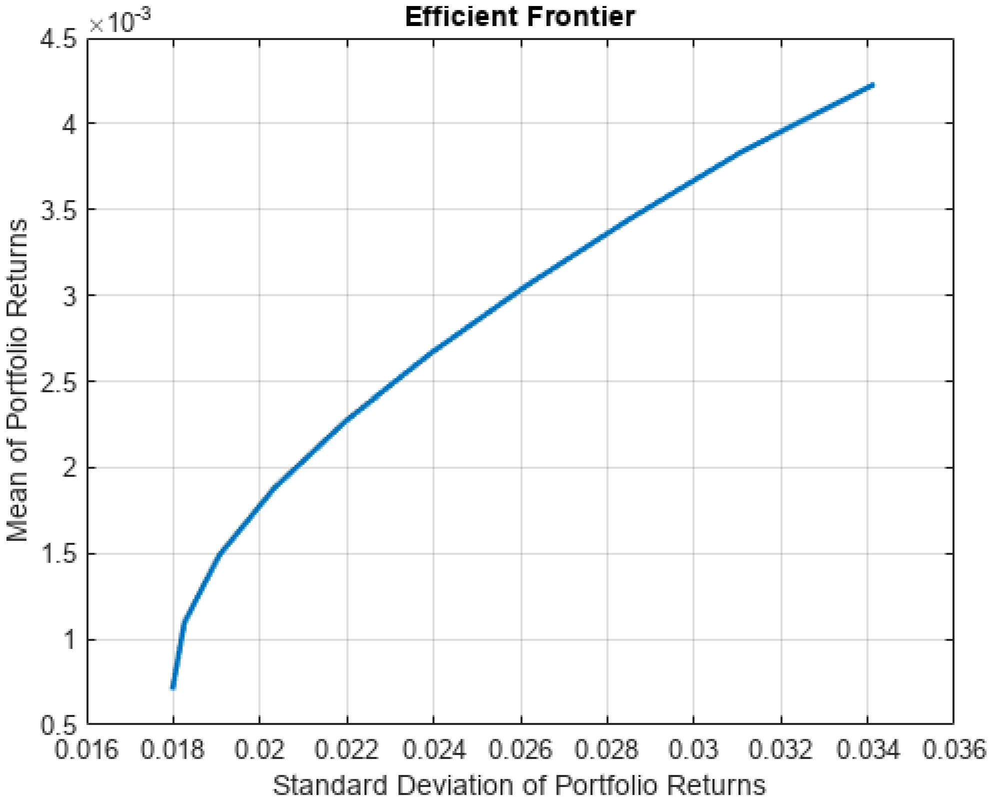
Examples

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 Portfolio Object with BoundType and MaxNumAssets Constraints

Create a `Portfolio` object for 12 stocks based on `CAPMuniverse.mat`.

```
load CAPMuniverse
p0 = Portfolio('AssetList',Assets(1:12));
p0 = estimateAssetMoments(p0, Data(:,1:12),'missingdata',true);
p0 = setDefaultConstraints(p0);
```

Use `setMinMaxNumAssets` to define a maximum number of 3 assets.

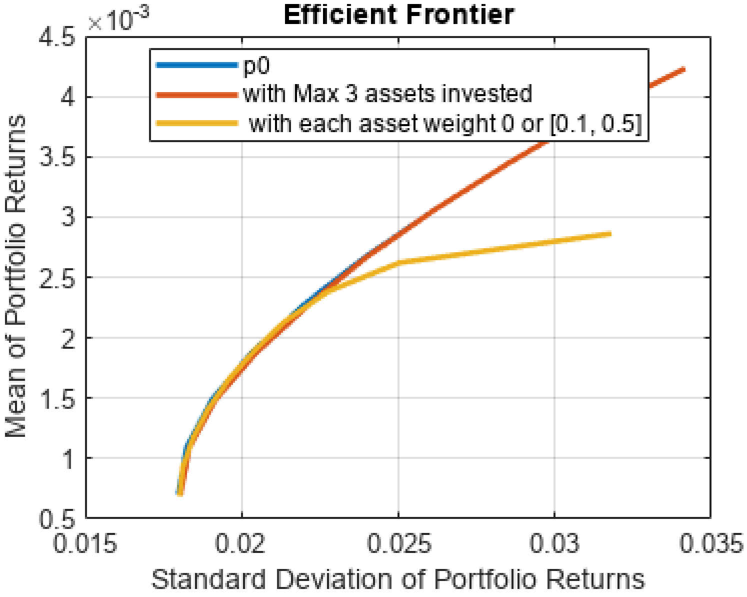
```
pWithMaxNumAssets = setMinMaxNumAssets(p0, [], 3);
```

Use `setBounds` to define a lower and upper bound and a `BoundType` of `'Conditional'`.

```
pWithConditionalBound = setBounds(p0, 0.1, 0.5,'BoundType', 'Conditional');
```

Use `plotFrontier` to compare the different portfolio objects.

```
figure;
plotFrontier(p0); hold on;
plotFrontier(pWithMaxNumAssets); hold on;
plotFrontier(pWithConditionalBound); hold off;
legend('p0', 'with Max 3 assets invested', ' with each asset weight 0 or [0.1, 0.5]', 'location'
```



Define a target return and use `estimateFrontierByReturn` to compare the three portfolio objects.

```
targetRetn = 2.0e-3;
pwgt0 = estimateFrontierByReturn(p0, targetRetn);
pwgtWithMaxNumAssets = estimateFrontierByReturn(pWithMaxNumAssets, targetRetn);
pwgtConditionalBound = estimateFrontierByReturn(pWithConditionalBound, targetRetn);
```

The following table shows the final allocation for specified target return among the three portfolio objects. You can see that the small positions in `'AAPL'`and `'HPQ'` are avoided in `pwgtConditionalBound`, and only three assets are invested in `pwgtWithMaxNumAssets`.

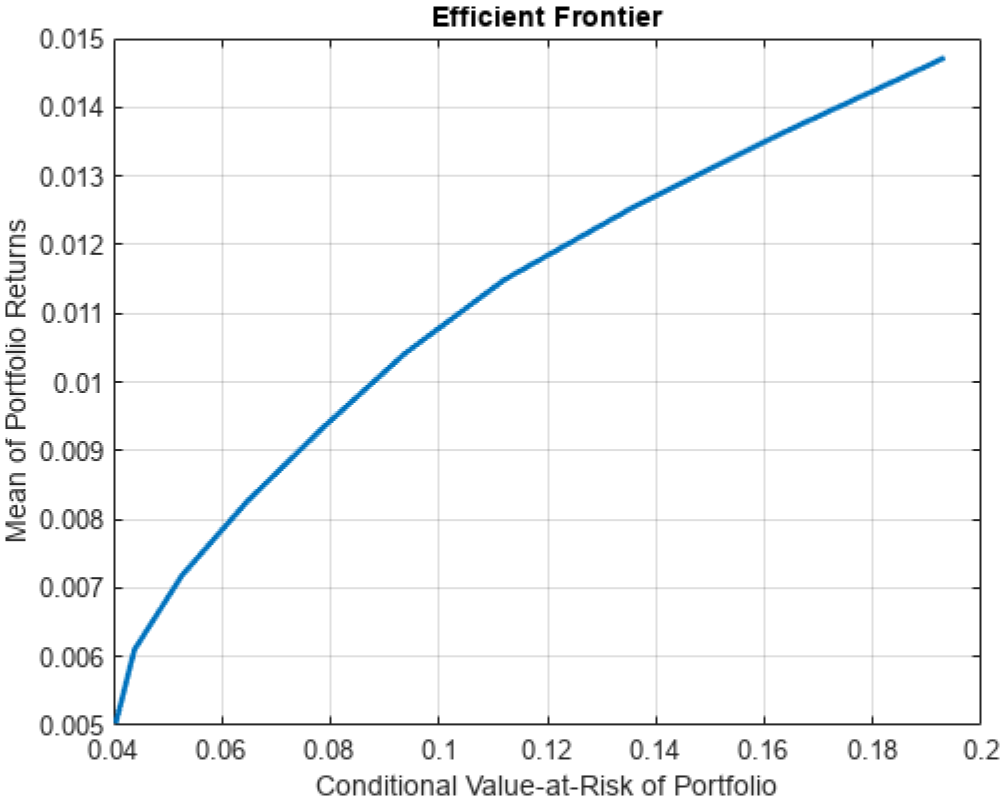
```
result = table(p0.AssetList',pwgt0,pwgtWithMaxNumAssets,pwgtConditionalBound)
```

```
result=12×4 table
      Var1         pwgt0       pwgtWithMaxNumAssets     pwgtConditionalBound
    ________    ________    ____________________    ____________________

    {'AAPL'}    0.076791        -3.0531e-16                   0.1
    {'AMZN'}           0                  0                     0
    {'CSCO'}           0                  0                     0
    {'DELL'}           0                  0                     0
    {'EBAY'}           0                  0                     0
    {'GOOG'}     0.44841            0.47297               0.44255
    {'HPQ' }    0.022406                  0                     0
    {'IBM' }     0.31139            0.34762               0.31592
    {'INTC'}           0                  0                     0
    {'MSFT'}     0.14101            0.17941               0.14153
    {'ORCL'}           0                  0                     0
    {'YHOO'}           0                  0                     0
```

Plot the Efficient Frontier for the PortfolioCVaR Object

Given a PortfolioCVaR p, plot the efficient frontier.

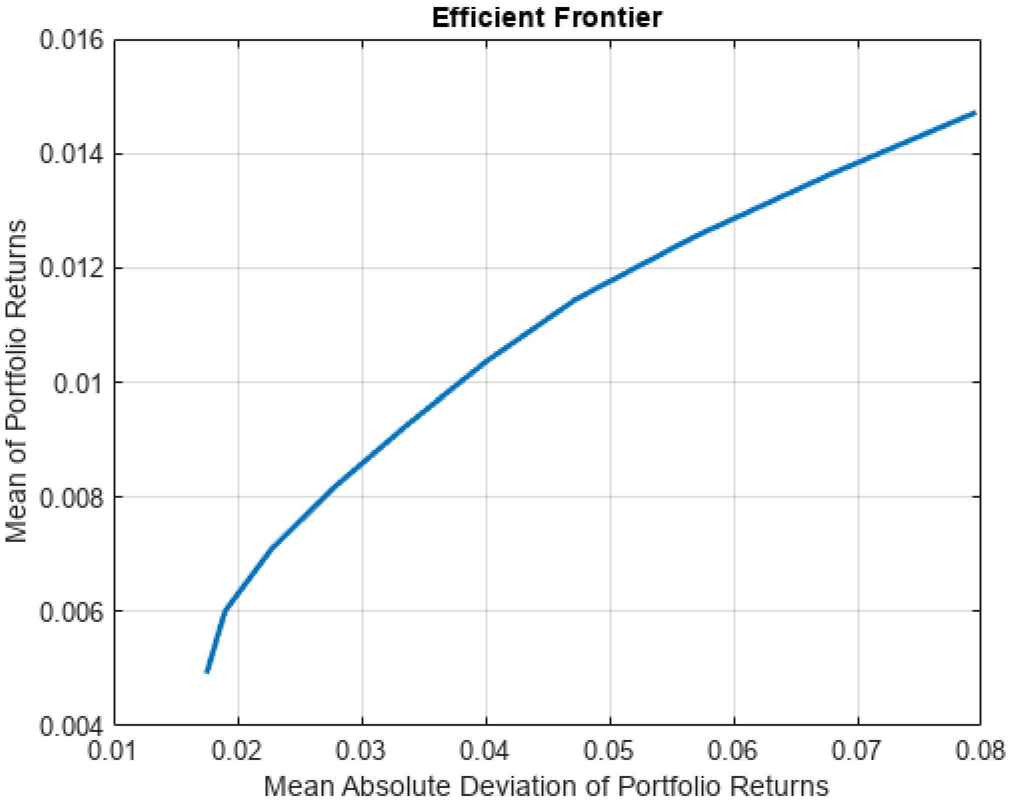
```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];
m = m/12;
C = C/12;

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);
```



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`

Data Types: `object`

`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`

`PortRisk` — Standard deviations of portfolio returns for each portfolio
`[]` (default) | vector

Standard deviations of portfolio returns for each portfolio, specified as a vector.

Note `PortRisk` and `PortReturn` must be vectors with the same size.

Data Types: `double`

`PortReturn` — Means of portfolio returns for each portfolio
`[]` (default) | vector

Means of portfolio returns for each portfolio, specified as a vector.

Note `PortRisk` and `PortReturn` must be vectors with the same size.

Data Types: `double`

`PortWeights` — Optimal portfolios on the efficient frontier
`[]` (default) | matrix

Optimal portfolios on the efficient frontier, specified as a `NumAsset`-by-`NumPorts` matrix.

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 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;
```

Version History

Introduced in R2011a

See Also

`estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierByRisk` | `estimateFrontierLimits`

Topics

“Plotting the Efficient Frontier for a Portfolio Object” on page 4-121
“Plotting the Efficient Frontier for a PortfolioCVaR Object” on page 5-105
“Plotting the Efficient Frontier for a PortfolioMAD Object” on page 6-100
“Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152

pointfig

Point and figure chart

Note `pointfig` is updated to accept data input as a matrix, `timetable`, or `table`.

Syntax

```
pointfig(Data)
h = pointfig(ax,Data)
```

Description

`pointfig(Data)` plots a point and figure chart from a series of prices of a security. Upward price movements are plotted as `X`'s and downward price movements are plotted as `O`'s.

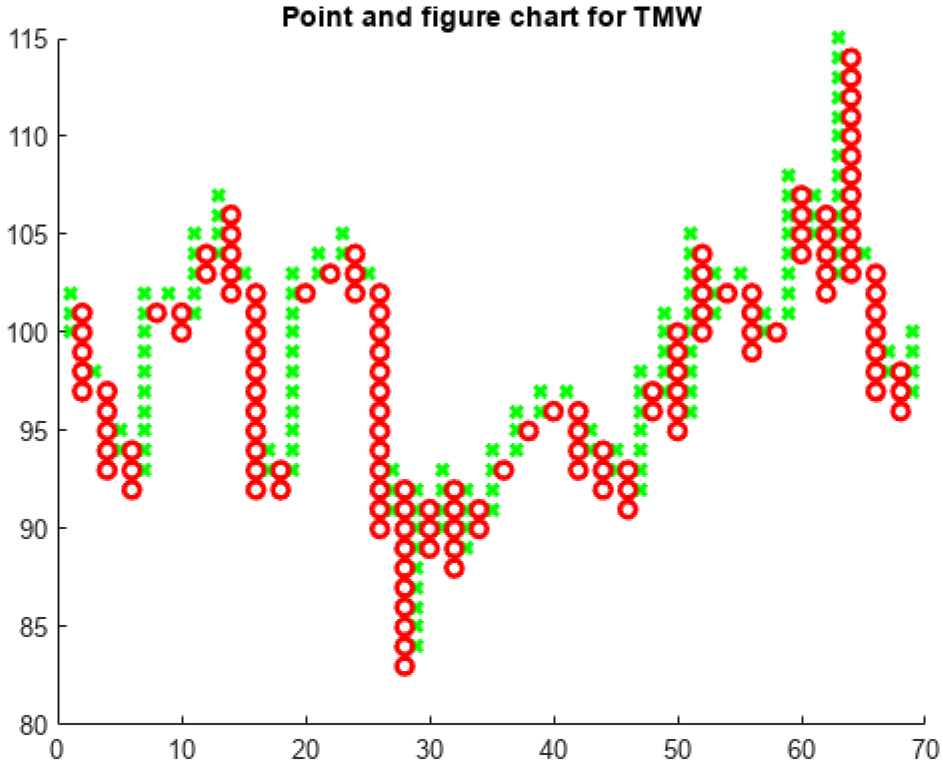
`h = pointfig(ax,Data)` adds an optional argument for `ax`.

Examples

Generate a Point and Figure Chart for a Data Series for a Stock

Load the file `SimulatedStock.mat`, which provides a timetable (`TMW`) for financial data for TMW stock. This Point and Figure chart is for closing prices of the stock TMW for the most recent 21 days. Note that the variable name of asset price is be renamed to `'Price'` (case insensitive).

```
load SimulatedStock.mat
TMW.Properties.VariableNames{'Close'} = 'Price';
pointfig(TMW(1:200,:))
title('Point and figure chart for TMW')
```



Input Arguments

Data — Data for a series of prices

matrix | table | timetable

Data for a series of prices, specified as a matrix, table, or timetable. Timetables and tables with M rows must contain a variable named `'Price'` (case insensitive).

Data Types: `double` | `table` | `timetable`

ax — Valid axis object

current axes (`ax = gca`) (default) | axes object

(Optional) Valid axis object, specified as an axes object. The point and figure plot is created in the axes specified by `ax` instead of in the current axes (`ax = gca`). The option `ax` can precede any of the input argument combinations.

Data Types: `object`

Output Arguments

h — Graphic handle of the figure

handle object

Graphic handle of the figure, returned as a handle object.

Version History

Introduced in R2006a

R2022b: Support for negative price data

Behavior changed in R2022b

The `Data` input accepts negative prices.

See Also

`timetable` | `table` | `movavg` | `linebreak` | `highlow` | `kagi` | `priceandvol` | `renko` | `volarea` | `candle`

Topics

"Use Timetables in Finance" on page 11-7
"Convert Financial Time Series Objects (fints) to Timetables" on page 11-2

portalloc

Optimal capital allocation to efficient frontier portfolios

Syntax

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

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

Description

`[RiskyRisk,RiskyReturn,RiskyWts,RiskyFraction,OverallRisk,OverallReturn] = portalloc(PortRisk,PortReturn,PortWts,RisklessRate)` calculates the optimal risky portfolio and the optimal allocation of funds between that risky portfolio of `NASSETS` and the risk-free asset.

Note An alternative for portfolio optimization is to use the `Portfolio` object 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 4-17.

`[RiskyRisk,RiskyReturn,RiskyWts,RiskyFraction,OverallRisk,OverallReturn] = portalloc( ___ ,BorrowRate,RiskAversion)` specifies options using one or more optional arguments in addition to the input arguments in the previous syntax.

`portalloc(PortRisk,PortReturn,PortWts,RisklessRate,BorrowRate,RiskAversion)` when invoked without any output arguments, a graph of the optimal capital allocation decision is displayed.

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 = 1×3

    0.0265    0.6023    0.3712


RiskyFraction = 1.1898

OverallRisk = 0.1527

OverallReturn = 0.1899
```

Input Arguments

PortRisk — Standard deviation of each risky asset efficient frontier portfolio
vector

Standard deviation of each risky asset efficient frontier portfolio, specified as an NPORTS-by-1 vector.

Data Types: double

PortReturn — Expected return of each risky asset efficient frontier portfolio
vector

Expected return of each risky asset efficient frontier portfolio, specified an NPORTS-by-1 vector.

Data Types: double

PortWts — Weights allocated to each asset
numeric

Weights allocated to each asset, specified as 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.

Data Types: double

RisklessRate — Risk-free lending rate
scalar decimal

Risk-free lending rate, specified as a scalar decimal.

Data Types: double

BorrowRate — Borrowing rate
NaN (default) | scalar decimal

(Optional) Borrowing rate, specified as a scalar decimal. If borrowing is not desired, or not an option, set the BorrowRate to NaN (which is the default value).

Data Types: double

RiskAversion — Coefficient of investor's degree of risk aversion
3 (default) | scalar numeric

(Optional) Coefficient of investor's degree of risk aversion, specified as a scalar numeric. Higher numbers indicate greater risk aversion. Typical coefficients range from 2.0 through 4.0. The default value of RiskAversion is 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.

Data Types: double

Output Arguments

RiskyRisk — Standard deviation of the optimal risky portfolio
numeric

Standard deviation of the optimal risky portfolio, returned as a scalar.

RiskyReturn — Expected return of the optimal risky portfolio
numeric

Expected return of the optimal risky portfolio, returned as a scalar.

RiskyWts — Weights allocated to the optimal risky portfolio
vector

Weights allocated to the optimal risky portfolio, returned a 1-by-NASSETS vector. The total of all weights in the portfolio is 1.

RiskyFraction — Fraction of the complete portfolio allocated to the risky portfolio
numeric

Fraction of the complete portfolio (that is, the overall portfolio including risky and risk-free assets) allocated to the risky portfolio, returned as a scalar.

OverallRisk — Standard deviation of the optimal overall portfolio
numeric

Standard deviation of the optimal overall portfolio, returned as a scalar.

OverallReturn — Expected rate of return of the optimal overall portfolio
numeric

Expected rate of return of the optimal overall portfolio, returned as a scalar.

Version History

Introduced before R2006a

References

[1] Bodie, Z., Kane, A., and A. Marcus. *Investments*. McGraw-Hill Education, 2013.

See Also

`portrand` | `portstats`

Topics

"Portfolio Construction Examples" on page 3-5
"Portfolio Optimization Functions" on page 3-3

portfolioRiskContribution

Compute individual asset risk contribution to overall portfolio volatility

Syntax

```
riskCont = portfolioRiskContribution(portWeights,Sigma)
riskCont = portfolioRiskContribution( ___ ,Name=Value)
```

Description

`riskCont = portfolioRiskContribution(portWeights,Sigma)` computes individual asset risk contribution to overall portfolio volatility and returns `riskCont` as a matrix of risk contributions.

`riskCont = portfolioRiskContribution( ___ ,Name=Value)` specifies an option using a name-value argument in addition to the input arguments in the previous syntax.

Examples

Compute Individual Asset Risk Contribution

Use `portfolioRiskContribution` to compute the risk contribution per asset with respect to the portfolio total risk.

Assume the returns covariance matrix is given by the following values.

```
Sigma = [0.0100    0.0075    0.0100    0.0150
         0.0075    0.0225    0.0150    0.0225
         0.0100    0.0150    0.0400    0.0450
         0.0150    0.0225    0.0450    0.0900];
```

The `portWeights` are the following values:

```
portWeights = [0.4101; 0.2734; 0.1899; 0.1266];
```

Use `portfolioRiskContribution` to compute the percent of risk contribution per asset.

```
riskCont = portfolioRiskContribution(portWeights,Sigma)
```

```
riskCont = 4×1

    0.2500
    0.2500
    0.2500
    0.2500
```

The default is to compute the relative risk contribution. However, you can compute the absolute risk contribution by using the name-value argument `RiskContributionType="absolute"`. Use `portfolioRiskContribution` to compute each asset's relative risk contribution.

```
riskCont = portfolioRiskContribution(portWeights,Sigma,RiskContributionType="relative")
riskCont = 4×1

    0.2500
    0.2500
    0.2500
    0.2500
```

Compute Individual Asset Risk Contribution When Returns Contain NaN Values

Use `portfolioRiskContribution` to compute the risk contribution of each asset with respect to the portfolio total risk when the assets returns used to compute the variance have `NaN` values.

Assume the returns covariance matrix is given by the following values.

```
load('CAPMuniverse.mat','AssetsTimeTable')
Sigma = cov(AssetsTimeTable{:,1:12},'partialrows')
```

```
Sigma = 12×12

    0.0012    0.0005    0.0005    0.0005    0.0005    0.0001    0.0004    0.0003    0.0006    0.0
    0.0005    0.0023    0.0007    0.0005    0.0009    0.0001    0.0005    0.0003    0.0006    0.0
    0.0005    0.0007    0.0012    0.0006    0.0007    0.0000    0.0006    0.0004    0.0007    0.0
    0.0005    0.0005    0.0006    0.0009    0.0006    0.0000    0.0005    0.0003    0.0006    0.0
    0.0005    0.0009    0.0007    0.0006    0.0017    0.0002    0.0005    0.0003    0.0005    0.0
    0.0001    0.0001    0.0000    0.0000    0.0002    0.0006   -0.0000    0.0000    0.0000    0.0
    0.0004    0.0005    0.0006    0.0005    0.0005   -0.0000    0.0009    0.0003    0.0005    0.0
    0.0003    0.0003    0.0004    0.0003    0.0003    0.0000    0.0003    0.0004    0.0003    0.0
    0.0006    0.0006    0.0007    0.0006    0.0005    0.0000    0.0005    0.0003    0.0010    0.0
    0.0003    0.0004    0.0005    0.0004    0.0004    0.0000    0.0003    0.0002    0.0005    0.0
      ⋮
```

The risk parity portfolio is subject to the following weights for all the assets.

```
portWeights = [0.1; 0.1; 0.1; 0.03; 0.1; 0.1; 0.1; 0.05; 0.1; 0.1; 0.1; 0.1];
```

Use `portfolioRiskContribution` to compute the percentage risk contribution per asset.

```
riskCont = portfolioRiskContribution(portWeights,Sigma)
```

```
riskCont = 12×1

    0.0911
    0.1261
    0.1103
    0.0251
    0.1197
    0.0203
    0.0805
    0.0258
    0.0978
    0.0660
      ⋮
```

The default values returned by portfolioRiskContribution are the relative risk contribution of the individual assets to the overall portfolio risk. Also, you can compute the absolute risk contribution by using the name-value argument RiskContributionType="absolute". Use portfolioRiskContribution to compute each asset's absolute risk contribution.

```
riskCont = portfolioRiskContribution(portWeights,Sigma,RiskContributionType="absolute")
```

```
riskCont = 12×1

    0.0023
    0.0031
    0.0027
    0.0006
    0.0030
    0.0005
    0.0020
    0.0006
    0.0024
    0.0016
      ⋮
```

Compute Individual Asset Risk Contribution When portWeights Is Matrix

Use portfolioRiskContribution to compute the risk contribution per asset with respect to the portfolio total risk when portWeights is a matrix.

Assume the returns covariance matrix is given by the following values.

```
Sigma = [0.0100    0.0075    0.0100    0.0150
         0.0075    0.0225    0.0150    0.0225
         0.0100    0.0150    0.0400    0.0450
         0.0150    0.0225    0.0450    0.0900];
```

The matrix of portWeights is the values:

```
portWeights = [0.25    0.10    0.10828;
               0.25    0.20    0.17197;
               0.25    0.30    0.28026;
               0.25    0.40    0.43949];
```

Use portfolioRiskContribution to compute the percent of risk contribution per asset.

```
riskCont = portfolioRiskContribution(portWeights,Sigma)
```

```
riskCont = 4×3

    0.1083    0.0308    0.0322
    0.1720    0.1005    0.0816
    0.2803    0.2735    0.2455
    0.4395    0.5952    0.6407
```

The default is to compute the relative risk contribution. However, you can compute the absolute risk contribution by using the name-value argument RiskContributionType="absolute". Use portfolioRiskContribution to compute each asset's absolute risk contribution.

```
riskCont = portfolioRiskContribution(portWeights,Sigma,RiskContributionType="absolute")
```

```
riskCont = 4×3

    0.0170    0.0060    0.0064
    0.0269    0.0194    0.0162
    0.0439    0.0528    0.0488
    0.0688    0.1149    0.1274
```

Input Arguments

portWeights — Portfolio weights
matrix

Portfolio weights, specified using an NumAssets-by-NumPortfolios matrix. portWeights must be nonempty, numeric, finite, and real. portWeights can be negative and does not need to sum to 1. portWeights and Sigma must have the same number of NumAssets.

Data Types: double

Sigma — Covariance matrix of returns
positive semidefinite covariance matrix

Covariance matrix of returns, specified using an NumAssets-by-NumAssets positive semidefinite covariance matrix. Sigma and portWeights must have the same number of NumAssets.

Note If Sigma is not a positive symmetric positive semidefinite matrix, use nearcorr to create a positive semidefinite matrix.

Data Types: double

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: riskCont = portfolioRiskContribution(portWeights,Sigma,RiskContributionType="relative")

RiskContributionType — Type of risk contribution to compute
"relative" (relative risk contribution) (default) | character vector with value of 'relative' or 'absolute' | string with value of "relative" or "absolute"

Type of risk contribution to compute, specified as RiskContributionType and a scalar character vector or string:

- "relative" — Computes the relative contribution.
- "absolute" — Computes the absolute risk contribution.

Data Types: char | string

Output Arguments

`riskCont` — Risk contributions
matrix

Risk contributions, returned as a matrix. The default values returned by `portfolioRiskContribution` are the relative contribution of the assets to the portfolio. However, `portfolioRiskContribution` can also return the absolute risk contribution by passing the name-value argument `RiskContributionType` with a value of `"absolute"`.

Version History

Introduced in R2022a

See Also

`riskBudgetingPortfolio`

Topics

"Risk Budgeting Portfolio" on page 4-280
"Backtest Using Risk-Based Equity Indexation" on page 4-285
"Create Hierarchical Risk Parity Portfolio" on page 4-291

riskBudgetingPortfolio

Compute risk budgeting portfolios

Syntax

```
pwgt = riskBudgetingPortfolio(Sigma)
pwgt = riskBudgetingPortfolio(Sigma,budget)
pwgt = riskBudgetingPortfolio( ___ ,Name=Value)
[pwgt,exitflag] = riskBudgetingPortfolio( ___ ,Name=Value)
```

Description

`pwgt = riskBudgetingPortfolio(Sigma)` computes risk budgeting portfolios for long-only, fully invested, risk budgeting portfolios. When a `budget` input argument is not provided, the budget is set the same for all assets.

A risk budgeting portfolio is an allocation strategy that focuses on the allocation of risk without considering the returns. The goal of this strategy is to ensure that each asset in a portfolio contributes a given target risk level to the overall portfolio volatility.

`pwgt = riskBudgetingPortfolio(Sigma,budget)` computes risk budgeting portfolios for the long-only, fully invested, risk budgeting portfolios associated with the target risk budgets (`Budget`).

`pwgt = riskBudgetingPortfolio( ___ ,Name=Value)` specifies options using name-value arguments in addition to the input arguments in the previous syntaxes.

`[pwgt,exitflag] = riskBudgetingPortfolio( ___ ,Name=Value)` add an output argument for `exitflag` and specifies options using name-value arguments in addition to the input arguments in the previous syntax.

Examples

Compute Risk Budgeting Portfolio

Use `riskBudgetingPortfolio` to compute long-only, fully invested, risk budgeting portfolios.

Assume the returns covariance matrix is given by the following values.

```
Sigma = [0.0100    0.0075    0.0100    0.0150
         0.0075    0.0225    0.0150    0.0225
         0.0100    0.0150    0.0400    0.0450
         0.0150    0.0225    0.0450    0.0900];
```

When a `budget` input argument is not provided, by default, the `riskBudgetingPortfolio` function computes a risk parity portfolio. The `riskBudgetingPortfolio` default is to consider that the risk contribution is the same for all assets and the weights are long-only and fully invested.

```
pwgt = riskBudgetingPortfolio(Sigma)
```

```
pwgt = 4×1
```

```
    0.4101
    0.2734
    0.1899
    0.1266
```

However, if a vector is passed for the optional input `budget`, the `riskBudgetingPortfolio` function computes the long-only, fully invested, risk budgeting portfolios that achieve the target risk contributions specified in `budget`.

```
pwgt = riskBudgetingPortfolio(Sigma, [0.1;0.2;0.3;0.4])
```

```
pwgt = 4×1

    0.2292
    0.2800
    0.2633
    0.2275
```

Compute Risk Budgeting Portfolio When Returns Contain NaN Values

Use `riskBudgetingPortfolio` to compute the risk budgeting portfolio when the asset returns contain `NaN` values.

Assume the returns covariance matrix is given by the following values.

```
load('CAPMuniverse.mat','AssetsTimeTable')
Sigma = cov(AssetsTimeTable{:,1:12},'partialrows')
```

```
Sigma = 12×12

    0.0012    0.0005    0.0005    0.0005    0.0005    0.0001    0.0004    0.0003    0.0006    0.0
    0.0005    0.0023    0.0007    0.0005    0.0009    0.0001    0.0005    0.0003    0.0006    0.0
    0.0005    0.0007    0.0012    0.0006    0.0007    0.0000    0.0006    0.0004    0.0007    0.0
    0.0005    0.0005    0.0006    0.0009    0.0006    0.0000    0.0005    0.0003    0.0006    0.0
    0.0005    0.0009    0.0007    0.0006    0.0017    0.0002    0.0005    0.0003    0.0005    0.0
    0.0001    0.0001    0.0000    0.0000    0.0002    0.0006   -0.0000    0.0000    0.0000    0.0
    0.0004    0.0005    0.0006    0.0005    0.0005   -0.0000    0.0009    0.0003    0.0005    0.0
    0.0003    0.0003    0.0004    0.0003    0.0003    0.0000    0.0003    0.0004    0.0003    0.0
    0.0006    0.0006    0.0007    0.0006    0.0005    0.0000    0.0005    0.0003    0.0010    0.0
    0.0003    0.0004    0.0005    0.0004    0.0004    0.0000    0.0003    0.0002    0.0005    0.0
      ⋮
```

If only the covariance matrix is provided, the `riskBudgetingPortfolio` default is to consider that the risk contribution is the same for all assets and the weights are long-only and fully invested. Basically, the default is to obtain the risk parity portfolio.

```
pwgt = riskBudgetingPortfolio(Sigma)
```

```
pwgt = 12×1

    0.0718
    0.0576
    0.0610
    0.0757
    0.0574
    0.1940
    0.0817
    0.1178
    0.0676
    0.0981
       ⋮
```

However, if a vector is passed for the optional input `budget`, then the risk budgeting portfolio is computed using the default `ccd` algorithm.

```
pwgt = riskBudgetingPortfolio(Sigma,[0.1; 0.04; 0.2; 0.05; 0.1; 0.1; 0.1; 0.05; 0.1; 0.03; 0.1;
```

```
pwgt = 12×1

    0.0871
    0.0315
    0.1415
    0.0482
    0.0717
    0.2235
    0.0995
    0.0757
    0.0829
    0.0386
       ⋮
```

Compute Risk Budgeting Portfolio When `budget` Is Matrix

This example shows how to use `riskBudgetingPortfolio` to compute the risk budgeting portfolio when the `budget` values are a matrix.

Assume the returns covariance matrix is given by the following values.

```
Sigma = [0.0100    0.0075    0.0100    0.0150;
         0.0075    0.0225    0.0150    0.0225;
         0.0100    0.0150    0.0400    0.0450;
         0.0150    0.0225    0.0450    0.0900];
```

If only the covariance matrix is provided, the `riskBudgetingPortfolio` default is to consider that the risk contribution is the same for all assets and the weights are long-only and fully invested. Basically, the default is to obtain the risk parity portfolio.

```
pwgt = riskBudgetingPortfolio(Sigma)
```

```
pwgt = 4×1

    0.4101
    0.2734
    0.1899
    0.1266
```

If a matrix is passed for the optional input `budget`, then the risk budgeting portfolio is computed.

```
B = [0.25    0.10    0.10828;
     0.25    0.20    0.17197;
     0.25    0.30    0.28026;
     0.25    0.40    0.43949];
pwgt = riskBudgetingPortfolio(Sigma,B)
```

```
pwgt = 4×3

    0.4101    0.2292    0.2500
    0.2734    0.2800    0.2500
    0.1899    0.2633    0.2500
    0.1266    0.2275    0.2500
```

Input Arguments

`Sigma` — Covariance matrix of returns

positive semidefinite covariance matrix

Covariance matrix of returns, specified using an `NumAssets`-by-`NumAssets` positive semidefinite covariance matrix.

`Sigma` and `budget` must have the same number of `NumAssets`.

Note If `Sigma` is not a symmetric positive semidefinite matrix, use `nearcorr` to create a positive semidefinite matrix.

Data Types: `double`

`budget` — Target risk budget for each asset

matrix

Target risk budget for each asset, specified using an `NumAssets`-by-`NumPortfolios` matrix. `budget` must be nonnegative and all columns must sum to `1` because the values for `budget` represent the percentage of risk that each asset contributes of the total portfolio risk.

`budget` and `Sigma` must have the same number of `NumAssets`.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `pwgt = riskBudgetingPortfolio(Sigma, [0.1;0.2;0.3;0.4],Algorithm="newton",BudgetTolerance=1e6)`

`Algorithm` — Optimization algorithm

`"ccd"` (default) | character vector with value `'ccd'` or `'newton'` | string with value `"ccd"` or `"newton"`

Optimization algorithm, specified as Algorithm and a character vector or string:

- "ccd" — Cyclical coordinate descent algorithm
- "newton" — Newton algorithm with backtracking line-search

Data Types: char | string

BudgetTolerance — Deviation tolerance from target budget
1e-8 (default) | positive numeric

Deviation tolerance from target budget, specified as BudgetTolerance and a scalar positive numeric.

Data Types: double

MaxIterations — Maximum number of iterations allowed for the Algorithm
1e6 for "ccd" or 1e3 for "newton" (default) | positive integer

Maximum number of iterations allowed for the Algorithm, specified as MaxIterations and a scalar positive integer.

Data Types: double

Output Arguments

pwgt — Portfolio weights
matrix

Portfolio weights, returned as a matrix in which each column represents a portfolio.

exitflag — Indication of why the algorithm stopped
vector

Indication of why the algorithm stopped, returned as a 1-by-NumPortfolios vector with one of the following values:

- 1 — The weights of the portfolio achieve the target risk budget up to BudgetTolerance accuracy.
- 0 — The number of iterations exceeded MaxIterations.
- -1 — The solver cannot find a descent direction. This value applies only to the "newton" option for the Algorithm.

.

Version History

Introduced in R2022a

See Also

portfolioRiskContribution

Topics

“Risk Budgeting Portfolio” on page 4-280

“Backtest Using Risk-Based Equity Indexation” on page 4-285
“Create Hierarchical Risk Parity Portfolio” on page 4-291

portalpha

Compute risk-adjusted alphas and returns for one or more assets

Syntax

```
portalpha(Asset,Benchmark)
portalpha(Asset,Benchmark,Cash)
[Alpha,RAReturn] = portalpha(Asset,Benchmark,Cash,Choice)
```

Description

`portalpha(Asset,Benchmark)` computes risk-adjusted alphas.

Note An alternative for portfolio optimization is to use the `Portfolio` object 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 4-17.

`portalpha(Asset,Benchmark,Cash)` computes risk-adjusted alphas using the optional argument `Cash`.

`[Alpha,RAReturn] = portalpha(Asset,Benchmark,Cash,Choice)` computes risk-adjusted alphas and returns for one or more methods specified by `Choice`.

Examples

Calculate the Risk-Adjusted Return

This example shows how to calculate the risk-adjusted return using `portalpha` and compare it with the fund and market's mean returns.

Use the example data with a fund, a market, and a cash series.

```
load FundMarketCash
Returns = tick2ret(TestData);
Fund = Returns(:,1);
Market = Returns(:,2);
Cash = Returns(:,3);
MeanFund = mean(Fund)
```

```
MeanFund = 0.0038
```

```
MeanMarket = mean(Market)
```

```
MeanMarket = 0.0030
```

```
[MM, aMM] = portalpha(Fund, Market, Cash, 'MM')
```

```
MM = 0.0022

aMM = 0.0052

[GH1, aGH1] = portalpha(Fund, Market, Cash, 'gh1')

GH1 = 0.0013

aGH1 = 0.0025

[GH2, aGH2] = portalpha(Fund, Market, Cash, 'gh2')

GH2 = 0.0022

aGH2 = 0.0052

[SML, aSML] = portalpha(Fund, Market, Cash, 'sml')

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.

Input Arguments

Asset — Asset returns

matrix

Asset returns, specified as a NUMSAMPLES x NUMSERIES matrix with NUMSAMPLES observations of asset returns for NUMSERIES asset return series.

Data Types: double

Benchmark — Returns for a benchmark asset

vector

Returns for a benchmark asset, specified as a 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.

Data Types: double

Cash — Riskless asset

0 (default) | numeric | vector

(Optional) Riskless asset, specified as a 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.

Data Types: double

Choice — Computed measures

'xs' (default) | character vector | cell array of character vectors

(Optional) Computed measures, specified as a character vector or cell array of character vectors to indicate one or more measures to be computed from among various 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:

Code	Description
`'xs'`	Excess Return (no risk adjustment)
`'sml'`	Security Market Line — The security market line shows that the relationship between risk and return is linear for individual securities (that is, increased risk = increased return).
`'capm'`	Jensen's Alpha — Risk-adjusted performance measure that represents the average return on a portfolio or investment, above or below that predicted by the capital asset pricing model (CAPM), given the portfolio's or investment's beta and the average market return.
`'mm'`	Modigliani & Modigliani — Measures the returns of an investment portfolio for the amount of risk taken relative to some benchmark portfolio.
`'gh1'`	Graham-Harvey 1 — Performance measure developed by John Graham and Campbell Harvey. The idea is to lever a fund's portfolio to exactly match the volatility of the S&P 500. The difference between the fund's levered return and the S&P 500 return is the performance measure.
`'gh2'`	Graham-Harvey 2 — In this measure, the idea is to lever up or down the fund's recommended investment strategy (using a Treasury bill), so that the strategy has the same volatility as the S&P 500.
`'all'`	Compute all measures.

`Choice` is specified by using the code from the table (for example, to select the Modigliani & Modigliani measure, `Choice` = `'mm'`). A single choice is either a 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.

Data Types: `char` | `cell`

Output Arguments

`Alpha` — Risk-adjusted alphas

matrix

Risk-adjusted alphas, returned as an `NUMCHOICES`-by-`NUMSERIES` matrix of risk-adjusted alphas for each series in `Asset` with each row corresponding to a specified measure in `Choice`.

`RAReturn` — Risk-adjusted returns

matrix

Risk-adjusted returns, returned as an `NUMCHOICES`-by-`NUMSERIES` matrix of risk-adjusted returns for each series in `Asset` with each row corresponding to a specified measure in `Choice`.

Note NaN values in the data are ignored and, if NaNs are present, some results could be unpredictable. Although the alphas are comparable across measures, risk-adjusted returns depend on whether the Asset or Benchmark is levered or unlevered to match its risk with the alternative. If Choice = 'all', the order of rows in Alpha and RAReturn follows the order in the table. In addition, Choice = 'all' overrides all other choices.

Version History

Introduced in R2006b

References

[1] Graham, J. R. 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.

[2] Lintner, J. "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.

[3] Modigliani, F. 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.

[4] Mossin, J. "Equilibrium in a Capital Asset Market." *Econometrica.* Vol. 34, No. 4, October 1966, pp. 768–783.

[5] Sharpe, W.F., "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 7-3
"Performance Metrics Overview" on page 7-2

portcons

Portfolio constraints

Syntax

```
ConSet = portcons(ConstType,consttype_values)
```

Description

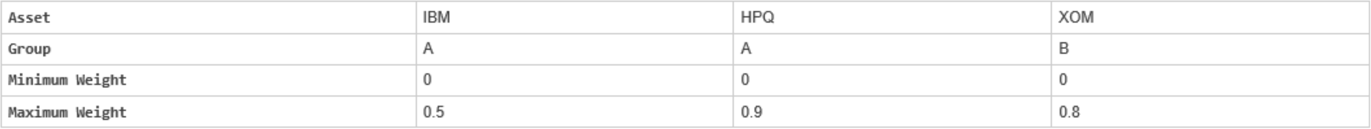
`ConSet = portcons(ConstType,consttype_values)` generates a matrix of constraints, using linear inequalities, for a portfolio of asset investments. The inequalities are of the type `A*Wts' <= b`, where `Wts` is the matrix of weights. The matrix `ConSet` is defined as `ConSet = [A b]`.

Note An alternative for portfolio optimization is to use the `Portfolio` object 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 4-17.

Examples

Generate Matrix of Constraints for Portfolio of Asset Investments

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
```

```
AtoBmax = 1.5000
```

```
              % in group B.
```

```
ConSet = portcons('PortValue', PVal, NumAssets,'AssetLims',...
AssetMin, AssetMax, NumAssets, 'GroupComparison',GroupA, NaN,...
AtoBmax, GroupB)
```

```
ConSet = 9×4

    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.

Input Arguments

ConstType — Constraint type

character vector with value 'Default', 'PortValue', 'AssetLims', 'GroupLims' ,'GroupComparison' , or 'Custom'

Constraint type, specified as a character vector defined as follows:

Constraint Type	Description	Values
'Default'	All allocations are >= 0; no short selling allowed. Combined value of portfolio allocations normalized to 1.	NumAssets (required). Scalar representing number of assets in portfolio.
'PortValue'	Fix total value of portfolio to PVal.	PVal (required). Scalar representing total value of portfolio. NumAssets (required). Scalar representing number of assets in portfolio. See pcpval.
'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 3-23.

Note You can specify multiple 'ConstType' arguments as ConSet = portcons('ConstType1',consttype_value1,'ConstType2',consttype_value2,'ConstTypeN',consttype_valueN).

Data Types: char

Output Arguments

ConSet — Constraints
matrix

Constraints, returned as a matrix. ConSet is defined as ConSet = [A b]. A is a matrix and b a vector such that A*Wts' <= b sets the value, where Wts is the matrix of weights.

Version History

Introduced before R2006a

See Also

pcalims | pcgcomp | pcglims | pcpval | portopt | Portfolio

Topics
“Portfolio Construction Examples” on page 3-5

“Portfolio Selection and Risk Aversion” on page 3-7
“Active Returns and Tracking Error Efficient Frontier” on page 3-25
“Analyzing Portfolios” on page 3-2
“Portfolio Optimization Functions” on page 3-3

Portfolio

Create Portfolio object for mean-variance portfolio optimization and analysis

Description

Use the `Portfolio` function to create a `Portfolio` object for 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. For details on this workflow, see “Portfolio Object Workflow” on page 4-17.

You can use the `Portfolio` object 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` object also accepts collections of name-value pair arguments for properties and their values. The `Portfolio` object 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` object to be an existing object with subsequent 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 15-1196). The `Portfolio` object 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 more detailed information on the theoretical basis for mean-variance optimization, see “Portfolio Optimization Theory” on page 4-3.

Creation

Syntax

```
p = Portfolio
```

```
p = Portfolio(Name,Value)
```

```
p = Portfolio(p,Name,Value)
```

Description

`p = Portfolio` creates 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 4-24.

`p = Portfolio(Name,Value)` creates a `Portfolio` object (`p`) and sets Properties on page 15-1182 using name-value pairs. For example, `p = Portfolio('AssetList',Assets(1:12))`. You can specify multiple name-value pairs.

`p = Portfolio(p,Name,Value)` creates a `Portfolio` object (`p`) using a previously created `Portfolio` object `p` and sets Properties on page 15-1182 using name-value pairs. You can specify multiple name-value pairs.

Input Arguments

p — Previously constructed Portfolio object
object

Previously constructed `Portfolio` object, specified using `Portfolio`.

Properties

Setting Up the Portfolio Object

`AssetList` — Names or symbols of assets in universe
`[]` (default) | cell array of character vectors | string array

Names or symbols of assets in the universe, specified as a cell array of character vectors or a string array.

Data Types: `cell` | `string`

`InitPort` — Initial portfolio
`[]` (default) | vector

Initial portfolio, specified as a vector.

Data Types: `double`

`Name` — Name for instance of Portfolio object
`[]` (default) | character vector | string

Name for instance of the `Portfolio` object, specified as a character vector or a string.

Data Types: `char` | `string`

`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. For more information, see “Linear Equality Constraints” on page 4-9.

Data Types: `double`

`AInequality` — Linear inequality constraint matrix

[ ] (default) | matrix

Linear inequality constraint matrix, specified as a matrix. For more information, see “Linear Inequality Constraints” on page 4-8.

Data Types: `double`

`bEquality` — Linear equality constraint vector

[ ] (default) | vector

Linear equality constraint vector, specified as a vector. For more information, see “Linear Equality Constraints” on page 4-9.

Data Types: `double`

`bInequality` — Linear inequality constraint vector

[ ] (default) | vector

Linear inequality constraint vector, specified as a vector. For more information, see “Linear Inequality Constraints” on page 4-8.

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. For more information, see “Group Constraints” on page 4-11.

Data Types: `double`

`GroupB` — Group B weights

[ ] (default) | matrix

Group B weights, specified as a matrix. For more information, see “Group Constraints” on page 4-11.

Data Types: `double`

`GroupMatrix` — Group membership matrix

[ ] (default) | matrix

Group membership matrix, specified as a matrix. For more information, see “Group Ratio Constraints” on page 4-12.

Data Types: `double`

`LowerBound` — Lower-bound constraint

[ ] (default) | vector

Lower-bound constraint, specified as a vector. For more information, see “'Simple' Bound Constraints” on page 4-9 and “'Conditional' Bound Constraints” on page 4-10.

Data Types: `double`

`LowerBudget` — Lower-bound budget constraint

`[ ]` (default) | scalar

Lower-bound budget constraint, specified as a scalar. For more information, see “Budget Constraints” on page 4-10.

Data Types: `double`

`LowerGroup` — Lower-bound group constraint

`[ ]` (default) | vector

Lower-bound group constraint, specified as a vector. For more information, see “Group Constraints” on page 4-11.

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. For more information, see “Group Ratio Constraints” on page 4-12.

Data Types: `double`

`TrackingError` — Upper bound for tracking error constraint

`[ ]` (default) | scalar

Upper bound for tracking error constraint, specified as a scalar. For more information, see “Tracking Error Constraints” on page 4-14.

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. For more information, see “'Simple' Bound Constraints” on page 4-9 and “'Conditional' Bound Constraints” on page 4-10.

Data Types: `double`

`UpperBudget` — Upper-bound budget constraint

`[ ]` (default) | scalar

Upper-bound budget constraint, specified as a scalar. For more information, see “Budget Constraints” on page 4-10.

Data Types: `double`

`UpperGroup` — Upper-bound group constraint

`[]` (default) | vector

Upper-bound group constraint, specified as a vector. For more information, see “Group Constraints” on page 4-11.

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. For more information, see “Group Ratio Constraints” on page 4-12.

Data Types: `double`

`BoundType` — Type of bounds for each asset weight

`'Simple'` (default) | character vector with value `'Simple'` or `'Conditional'` | string with value `"Simple"` or `"Conditional"` | cell array of character vectors with values `'Simple'` or `'Conditional'` | string array with values `"Simple"` or `"Conditional"`

Type of bounds for each asset weight, specified as a scalar character vector or string, or a cell array of character vectors or a string array. For more information, see `setBounds`.

Data Types: `char` | `cell` | `string`

`MinNumAssets` — Minimum number of assets allocated in portfolio

`[]` (default) | numeric

Minimum number of assets allocated in portfolio, specified as a scalar numeric value. For more information, see `setMinMaxNumAssets` and “Cardinality Constraints” on page 4-14.

Data Types: `double`

`MaxNumAssets` — Maximum number of assets allocated in portfolio

`[]` (default) | numeric

Maximum number of assets allocated in portfolio, specified as a scalar numeric value. For more information, see `setMinMaxNumAssets` and “Cardinality Constraints” on page 4-14.

Data Types: `double`

`SellTurnover` — Turnover constraint on sales

`[]` (default) | scalar

Turnover constraint on sales, specified as a scalar. For more information, see “Average Turnover Constraints” on page 4-12 and “One-Way Turnover Constraints” on page 4-13.

Data Types: `double`

`BuyTurnover` — Turnover constraint on purchases

`[]` (default) | scalar

Turnover constraint on purchases, specified as a scalar. For more information, see “Average Turnover Constraints” on page 4-12 and “One-Way Turnover Constraints” on page 4-13.

Data Types: `double`

`Turnover` — Turnover constraint

[ ] (default) | scalar

Turnover constraint, specified as a scalar. For more information, see “Average Turnover Constraints” on page 4-12 and “One-Way Turnover Constraints” on page 4-13.

Data Types: `double`

Portfolio Object Modeling

`AssetCovar` — Covariance of asset returns

[ ] (default) | square matrix

Covariance of asset returns, specified as a square matrix.

- If `AssetCovar` is not a symmetric positive semidefinite matrix, use `nearcorr` to create a positive semidefinite matrix for a correlation matrix.
- In some situations, the traditional sample approximation of the covariance matrix performs poorly. An intuitive example is when the number of variables is larger than the number of observations; this results in a noninvertible covariance matrix. Furthermore, because covariance estimation is performed using observations from random data, the estimator contains a certain amount of noise. This noise usually results in ill-conditioned covariance estimates. Working with an ill-conditioned matrix magnifies the impact of estimation errors. You can use `covarianceShrinkage` and `covarianceDenoising` for covariance estimation and noise reduction.

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. For more information, see “Net Portfolio Returns” on page 4-4.

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. For more information, see “Net Portfolio Returns” on page 4-4.

Data Types: `double`

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 for portfolio
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 for portfolio
setBudget	Set up budget constraints for portfolio
setCosts	Set up proportional transaction costs for portfolio
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
setMinMaxNumAssets	Set cardinality constraints on the number of assets invested in a portfolio
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
estimateCustomObjectivePortfolio	Estimate optimal portfolio for user-defined objective function for Portfolio object
setSolver	Choose main solver and specify associated solver options for portfolio optimization
setSolverMINLP	Choose mixed integer nonlinear programming (MINLP) solver for portfolio optimization

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: []
     MinNumAssets: []
     MaxNumAssets: []
        BoundType: []
```

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 object directly to 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)
```

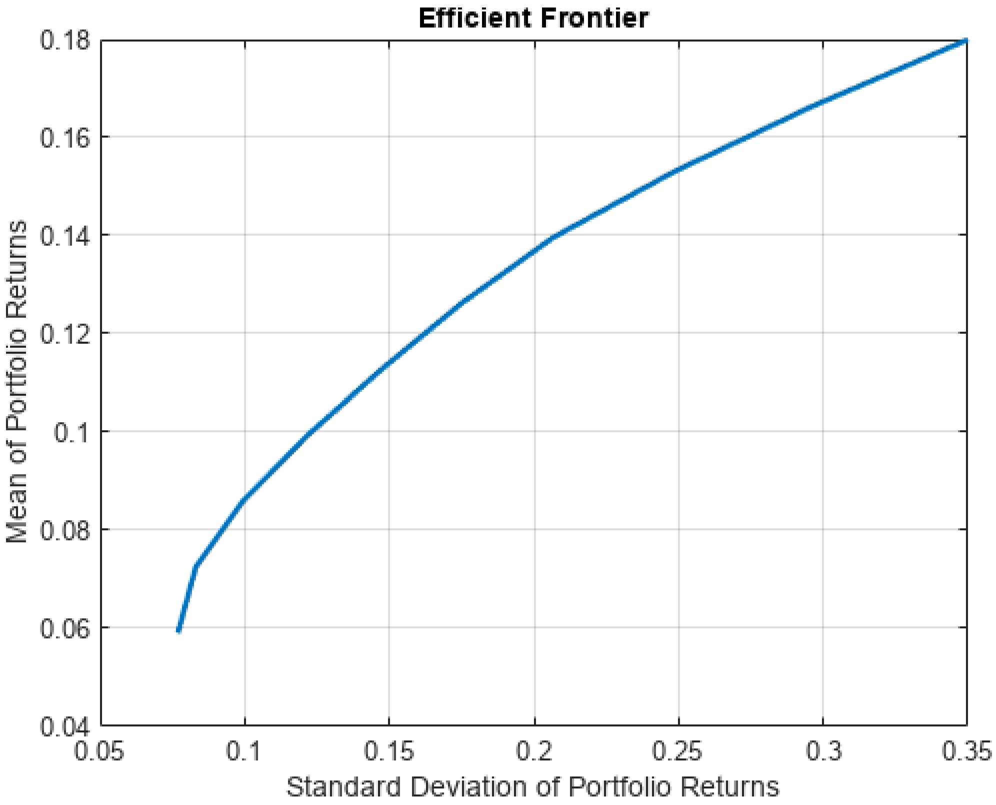
```
p = 
  Portfolio with properties:

          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: []
     MinNumAssets: []
     MaxNumAssets: []
        BoundType: []
```

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

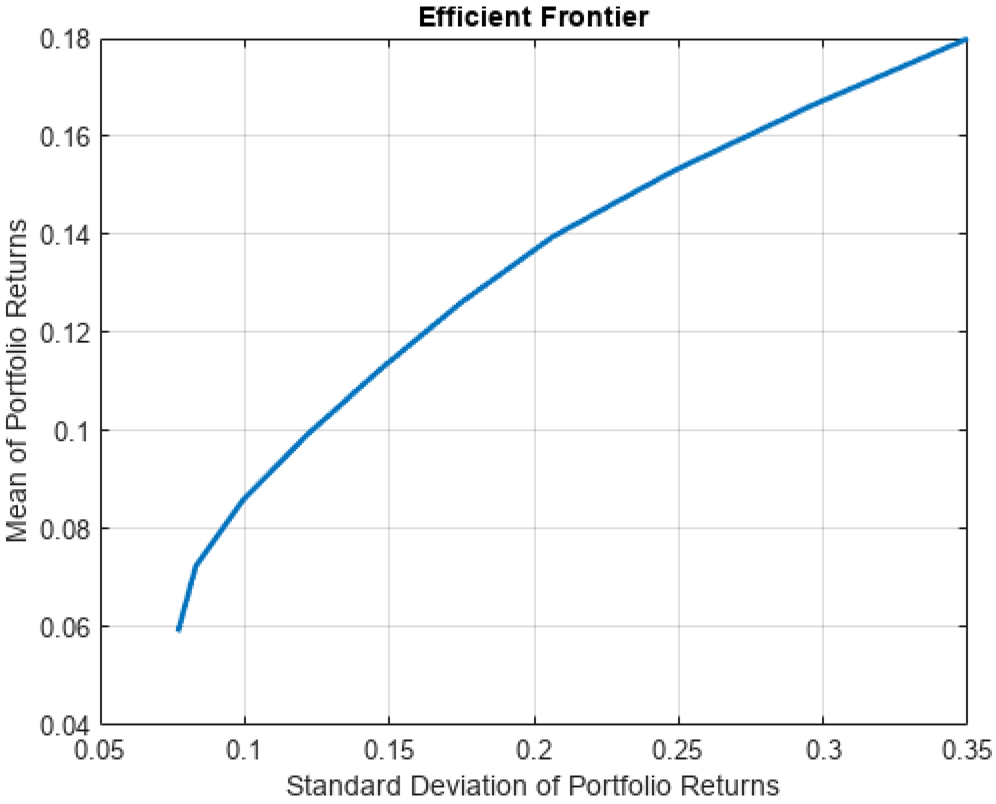
Using a sequence of steps is an alternative way to accomplish the same task of setting up a “standard” portfolio optimization problem, given a mean and covariance of asset returns in the variables m and C (which also illustrates that argument names are not case sensitive).

```
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 Portfolio 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` object 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 `Portfolio` 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: []
     MinNumAssets: []
     MaxNumAssets: []
        BoundType: []
```

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;
```

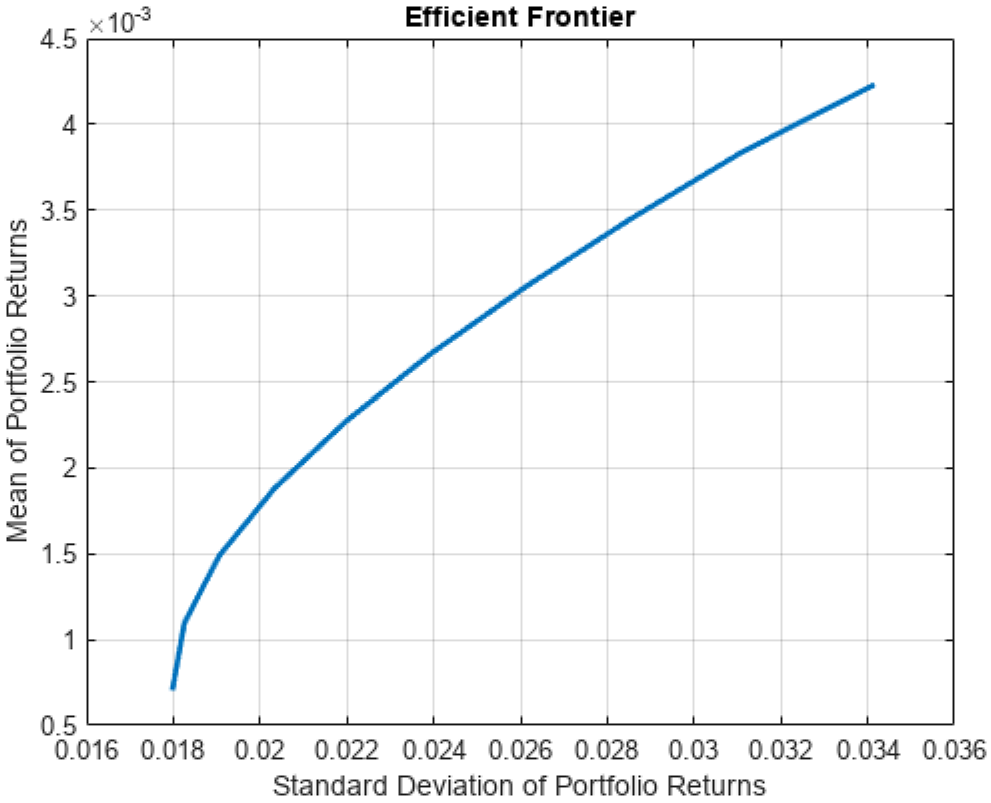
```
p.AssetCovar = C;
p.LowerBudget = 1;
p.UpperBudget = 1;
p.LowerBound = zeros(size(m));
disp(p)
```

```
  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: []
     MinNumAssets: []
     MaxNumAssets: []
        BoundType: []
```

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
```

More About

Mean-Variance Portfolio Optimization

A mean-variance optimization problem is completely specified with three elements.

The three elements for a mean-variance optimization problem are:

- 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

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 `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.

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);
```

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-5.

Shortcuts for Property Names

The `Portfolio` object 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` object 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` object. 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`

Version History

Introduced in R2011a

References

[1] For a complete list of references for the Portfolio object, see “Portfolio Optimization” on page A-5.

See Also

`plotFrontier` | `estimateFrontier` | `PortfolioCVaR` | `PortfolioMAD` | `nearcorr` | `covarianceShrinkage` | `covarianceDenoising`

Topics

“Creating the Portfolio Object” on page 4-24
“Working with Portfolio Constraints Using Defaults” on page 4-57
“Estimate Efficient Portfolios for Entire Efficient Frontier for Portfolio Object” on page 4-94
“Estimate Efficient Frontiers for Portfolio Object” on page 4-118
“Asset Allocation Case Study” on page 4-172
“Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
“Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-183
“Black-Litterman Portfolio Optimization Using Financial Toolbox™” on page 4-215
“Portfolio Optimization Using Factor Models” on page 4-224
“Bond Portfolio Optimization Using Portfolio Object” on page 10-30
“Portfolio Optimization Theory” on page 4-3
“Portfolio Object Workflow” on page 4-17
“Portfolio Object Properties and Functions” on page 4-19
“Working with Portfolio Objects” on page 4-19
“Setting and Getting Properties” on page 4-19
“Displaying Portfolio Objects” on page 4-20
“Saving and Loading Portfolio Objects” on page 4-20
“Estimating Efficient Portfolios and Frontiers” on page 4-20
“Arrays of Portfolio Objects” on page 4-21
“Subclassing Portfolio Objects” on page 4-22
“Conventions for Representation of Data” on page 4-22
“Portfolio Set for Optimization Using Portfolio Objects” on page 4-8
“Role of Convexity in Portfolio Problems” on page 4-148
“When to Use Portfolio Objects Over Optimization Toolbox” on page 4-128
“Choosing and Controlling the Solver for Mean-Variance Portfolio Optimization” on page 4-110

External Websites

Getting Started with Portfolio Optimization (4 min 12 sec)

PortfolioCVaR

Creates PortfolioCVaR object for conditional value-at-risk portfolio optimization and analysis

Description

Use `PortfolioCVaR` to create a `PortfolioCVaR` object for conditional value-at-risk 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. For details on this workflow, see “PortfolioCVaR Object Workflow” on page 5-16.

You can use the `PortfolioCVaR` object 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` object also accepts collections of 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` object to be an existing object with subsequent 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 15-1213). The `PortfolioCVaR` object 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 more detailed information on the theoretical basis for conditional value-at-risk portfolio optimization, see “Portfolio Optimization Theory” on page 5-3.

Creation

Syntax

```
p = PortfolioCVaR
p = PortfolioCVaR(Name,Value)

p = PortfolioCVaR(p,Name,Value)
```

Description

`p = PortfolioCVaR` creates 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 5-22.

`p = PortfolioCVaR(Name,Value)` creates a `PortfolioCVaR` object (`p`) and sets Properties on page 15-1199 using name-value pairs. For example, `p = PortfolioCVaR('AssetList',Assets(1:12))`. You can specify multiple name-value pairs.

`p = PortfolioCVaR(p,Name,Value)` creates a `PortfolioCVaR` object (`p`) using a previously created `PortfolioCVaR` object `p` and sets Properties on page 15-1199 using name-value pairs. You can specify multiple name-value pairs.

Input Arguments

p — Previously constructed PortfolioCVaR object
object

Previously constructed `PortfolioCVaR` object, specified using `PortfolioCVaR`.

Properties

Setting Up the PortflioCVaR Object

`AssetList` — Names or symbols of assets in universe
`[ ]` (default) | cell array of character vectors | string array

Names or symbols of assets in the universe, specified as a cell array of character vectors or a string array.

Data Types: `cell` | `string`

`InitPort` — Initial portfolio
`[ ]` (default) | vector

Initial portfolio, specified as a vector.

Data Types: `double`

`Name` — Name for instance of PortfolioCVaR object
`[ ]` (default) | character vector | string

Name for instance of the `PortfolioCVaR` object, specified as a character vector or string.

Data Types: `char` | `string`

`NumAssets` — Number of assets in the universe

`[ ]` (default) | integer scalar

Number of assets in the universe, specified as an integer scalar.

Data Types: `double`

PortfolioCVaR Object Constraints

`AEquality` — Linear equality constraint matrix

`[ ]` (default) | matrix

Linear equality constraint matrix, specified as a matrix. For more information, see “Linear Equality Constraints” on page 5-9.

Data Types: `double`

`AInequality` — Linear inequality constraint matrix

`[ ]` (default) | matrix

Linear inequality constraint matrix, specified as a matrix. For more information, see “Linear Inequality Constraints” on page 5-8.

Data Types: `double`

`bEquality` — Linear equality constraint vector

`[ ]` (default) | vector

Linear equality constraint vector, specified as a vector. For more information, see “Linear Equality Constraints” on page 5-9.

Data Types: `double`

`bInequality` — Linear inequality constraint

`[ ]` (default) | vector

Linear inequality constraint vector, specified as a vector. For more information, see “Linear Inequality Constraints” on page 5-8.

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. For more information, see “Group Constraints” on page 5-11.

Data Types: `double`

`GroupB` — Group B weights

`[ ]` (default) | matrix

Group B weights, specified as a matrix. For more information, see “Group Constraints” on page 5-11.

Data Types: `double`

GroupMatrix — Group membership matrix

[ ] (default) | matrix

Group membership matrix, specified as a matrix. For more information, see "Group Ratio Constraints" on page 5-11.

Data Types: double

LowerBound — Lower-bound constraint

[ ] (default) | vector

Lower-bound constraint, specified as a vector. For more information, see "'Simple' Bound Constraints" on page 5-9 and "'Conditional' Bound Constraints" on page 5-10.

Data Types: double

LowerBudget — Lower-bound budget constraint

[ ] (default) | scalar

Lower-bound budget constraint, specified as a scalar. For more information, see "Budget Constraints" on page 5-10.

Data Types: double

LowerGroup — Lower-bound group constraint

[ ] (default) | vector

Lower-bound group constraint, specified as a vector. For more information, see "Group Constraints" on page 5-11.

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. For more information, see "Group Ratio Constraints" on page 5-11.

Data Types: double

UpperBound — Upper-bound constraint

[ ] (default) | vector

Upper-bound constraint, specified as a vector. For more information, see "'Simple' Bound Constraints" on page 5-9 and "'Conditional' Bound Constraints" on page 5-10.

Data Types: double

UpperBudget — Upper-bound budget constraint

[ ] (default) | scalar

Upper-bound budget constraint, specified as a scalar. For more information, see "Budget Constraints" on page 5-10.

Data Types: double

UpperGroup — Upper-bound group constraint

[ ] (default) | vector

Upper-bound group constraint, specified as a vector. For more information, see “Group Constraints” on page 5-11.

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. For more information, see “Group Ratio Constraints” on page 5-11.

Data Types: double

BoundType — Type of bounds for each asset weight

'Simple' (default) | character vector with value 'Simple' or 'Conditional' | string with value "Simple" or "Conditional" | cell array of character vectors with values 'Simple' or 'Conditional' | string array with values "Simple" or "Conditional"

Type of bounds for each asset weight, specified as a scalar character vector or string, or a cell array of character vectors or a string array. For more information, see setBounds.

Data Types: char | cell | string

MinNumAssets — Minimum number of assets allocated in portfolio

[ ] (default) | numeric

Minimum number of assets allocated in portfolio, specified as a scalar numeric value. For more information, see setMinMaxNumAssets and “Cardinality Constraints” on page 5-13.

Data Types: double

MaxNumAssets — Maximum number of assets allocated in portfolio

[ ] (default) | numeric

Maximum number of assets allocated in portfolio, specified as a scalar numeric value. For more information, see setMinMaxNumAssets and “Cardinality Constraints” on page 5-13.

Data Types: double

Turnover — Turnover constraint

[ ] (default) | scalar

Turnover constraint, specified as a scalar. For more information, see “Average Turnover Constraints” on page 5-12 and “One-way Turnover Constraints” on page 5-13.

Data Types: double

SellTurnover — Turnover constraint on sales

[ ] (default) | scalar

Turnover constraint on sales, specified as a scalar. For more information, see “Average Turnover Constraints” on page 5-12 and “One-way Turnover Constraints” on page 5-13.

Data Types: double

BuyTurnover — Turnover constraint on purchases

[ ] (default) | scalar

Turnover constraint on purchases, specified as a scalar. For more information, see “Average Turnover Constraints” on page 5-12 and “One-way Turnover Constraints” on page 5-13.

Data Types: `double`

PortfolioCVaR Object Modeling

`BuyCost` — Proportional purchase costs

`[]` (default) | vector

Proportional purchase costs, specified as a vector. For more information, see “Net Portfolio Returns” on page 5-4.

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. For more information, see “Net Portfolio Returns” on page 5-4.

Data Types: `double`

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 for portfolio
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 for portfolio
setMinMaxNumAssets	Set cardinality constraints on the number of assets invested in a portfolio
setBudget	Set up budget constraints for portfolio
setCosts	Set up proportional transaction costs for portfolio
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	Estimate standard deviation of portfolio returns

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: []
        MinNumAssets: []
        MaxNumAssets: []
           BoundType: []
```

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` object directly to 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)
```

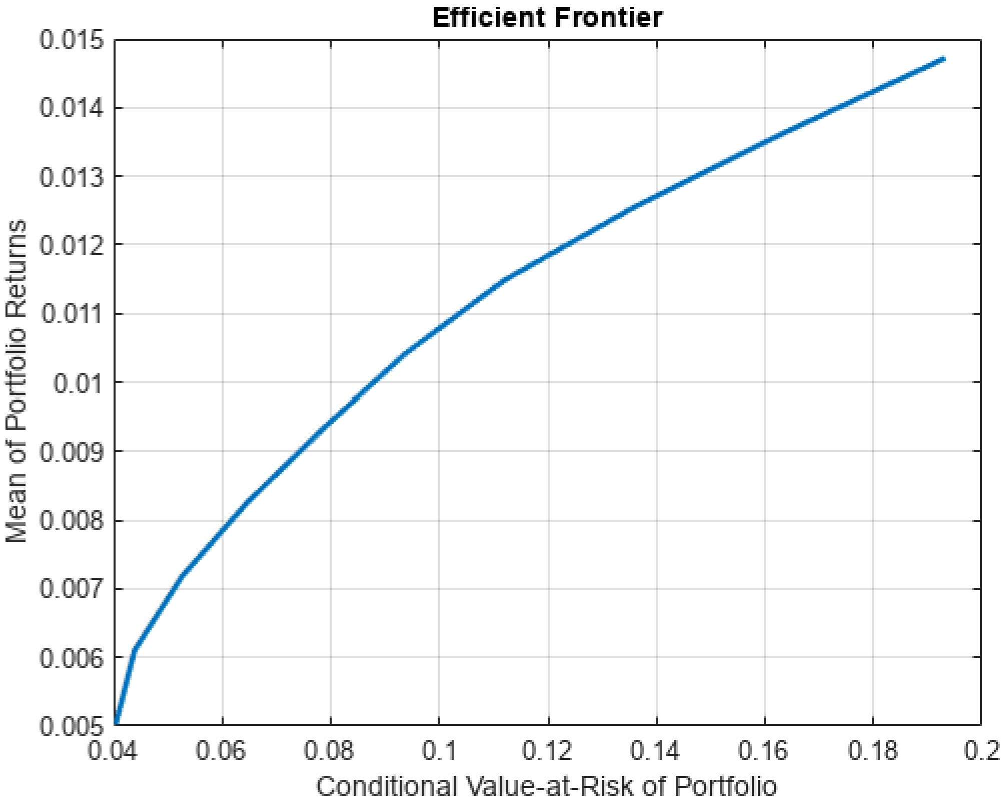
```
p = 
  PortfolioCVaR with properties:

             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: []
        MinNumAssets: []
        MaxNumAssets: []
           BoundType: []
```

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

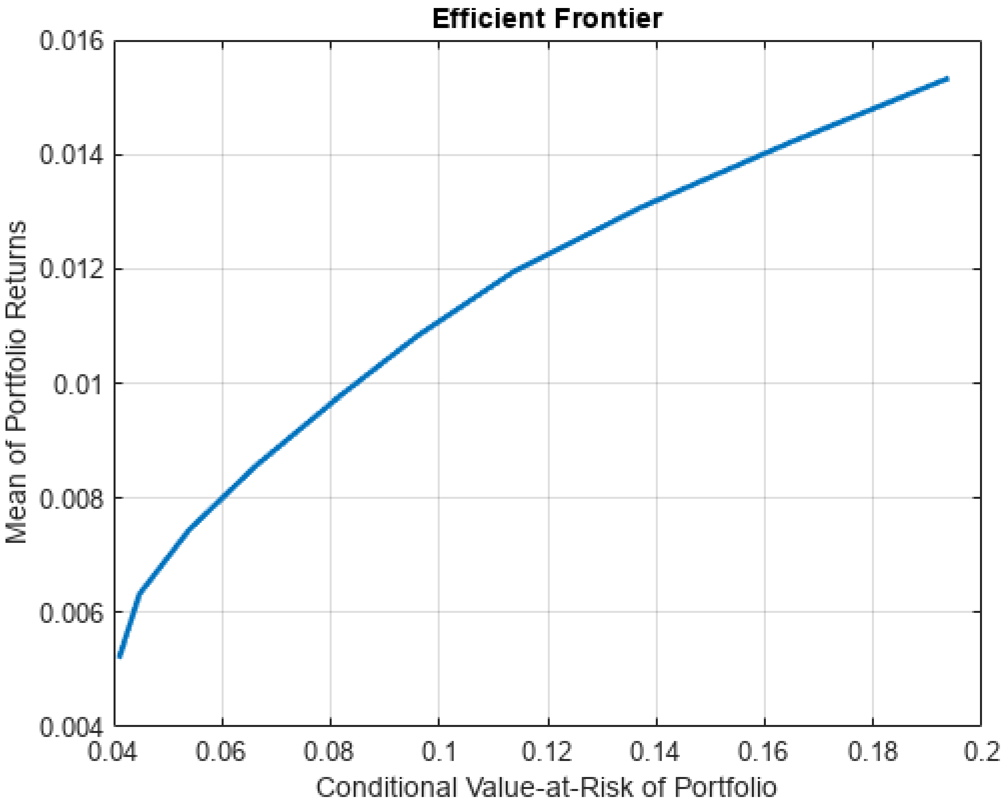
Using a sequence of steps is an alternative way to accomplish the same task of setting up a “standard” CVaR portfolio optimization problem, given `AssetScenarios` variable is:

```
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 `PortfolioCVaR` 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` object 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` object 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: []
        MinNumAssets: []
        MaxNumAssets: []
           BoundType: []
```

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)
```

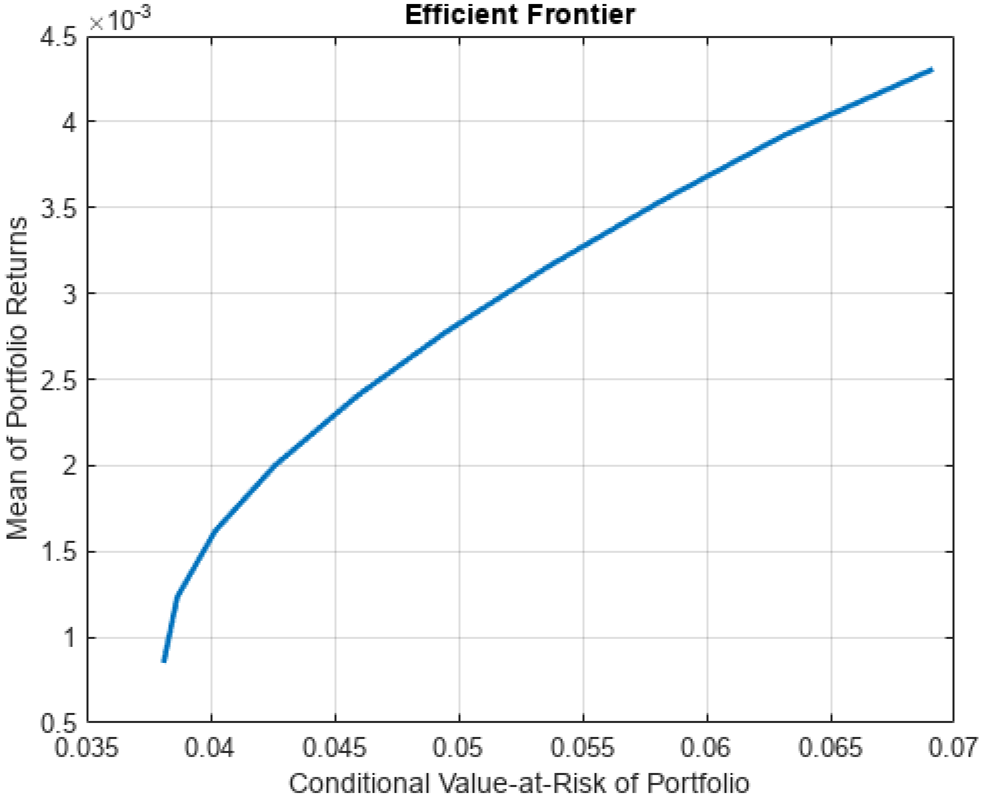
```
  PortfolioCVaR with properties:

             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: []
        MinNumAssets: []
        MaxNumAssets: []
           BoundType: []
```

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    0.011002       0.07341       0.11855    0.12957     0
    AMZN           0             0             0          0     0
    CSCO           0            0            0           0    0
    DELL    0.023234            0            0           0    0
    EBAY           0            0            0           0    0
    GOOG     0.20304       0.3804      0.56259     0.75956    1
    HPQ     0.041781    0.0094108            0           0    0
    IBM       0.4452      0.36408       0.2625     0.11086    0
    INTC           0            0            0           0    0
    MSFT     0.27575       0.1727     0.056365           0    0
    ORCL           0            0            0           0    0
    YHOO           0            0            0           0    0
```

More About

Conditional Value-at-Risk Portfolio Optimization

A CVaR optimization problem is completely specified with four elements.

The four elements for a CVaR optimization problem are:

- 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 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` object 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` object 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` object. 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

Version History

Introduced in R2012b

References

[1] For a complete list of references for the PortfolioCVaR object, see "Portfolio Optimization" on page A-5.

See Also

plotFrontier | estimateFrontier | setScenarios | Portfolio | PortfolioMAD | nearcorr

Topics

"Creating the PortfolioCVaR Object" on page 5-22
"Common Operations on the PortfolioCVaR Object" on page 5-29
"Working with CVaR Portfolio Constraints Using Defaults" on page 5-50
"Asset Returns and Scenarios Using PortfolioCVaR Object" on page 5-36
"Estimate Efficient Portfolios for Entire Frontier for PortfolioCVaR Object" on page 5-82
"Estimate Efficient Frontiers for PortfolioCVaR Object" on page 5-102
"Postprocessing Results to Set Up Tradable Portfolios" on page 5-110
"Hedging Using CVaR Portfolio Optimization" on page 5-118
"Portfolio Optimization Theory" on page 5-3
"PortfolioCVaR Object Workflow" on page 5-16
"PortfolioCVaR Object Properties and Functions" on page 5-17
"Working with PortfolioCVaR Objects" on page 5-17
"Setting and Getting Properties" on page 5-18
"Displaying PortfolioCVaR Objects" on page 5-18
"Saving and Loading PortfolioCVaR Objects" on page 5-18
"Estimating Efficient Portfolios and Frontiers" on page 5-18

“Arrays of PortfolioCVaR Objects” on page 5-19
“Subclassing PortfolioCVaR Objects” on page 5-20
“Conventions for Representation of Data” on page 5-20
“Portfolio Set for Optimization Using PortfolioCVaR Object” on page 5-8
“Choosing and Controlling the Solver for PortfolioCVaR Optimizations” on page 5-95

External Websites
Getting Started with Portfolio Optimization (4 min 13 sec)
CVaR Portfolio Optimization (4 min 56 sec)

PortfolioMAD

Create 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. For more information on the workflow when using PortfolioMAD objects, see “PortfolioMAD Object Workflow” on page 6-15.

You can use the PortfolioMAD object 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 object also accepts collections of name-value pair arguments for properties and their values. The PortfolioMAD object 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 object to be an existing object with subsequent 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 15-1231). The PortfolioMAD object 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 more detailed information on the theoretical basis for conditional value-at-risk portfolio optimization, see “Portfolio Optimization Theory” on page 6-2.

Creation

Syntax

```
p = PortfolioMAD
p = PortfolioMAD(Name,Value)

p = PortfolioMAD(p,Name,Value)
```

Description

`p = PortfolioMAD` creates 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 6-21.

`p = PortfolioMAD(Name,Value)` creates a `PortfolioMAD` object (`p`) and sets Properties on page 15-1217 using name-value pairs. For example, `p = PortfolioMAD('AssetList',Assets(1:12))`. You can specify multiple name-value pairs.

`p = PortfolioMAD(p,Name,Value)` creates a `PortfolioMAD` object (`p`) using a previously created `PortfolioMAD` object `p` and sets Properties on page 15-1217 using name-value pairs. You can specify multiple name-value pairs.

Input Arguments

p — Previously constructed PortfolioMAD object
object

Previously constructed `PortfolioMAD` object, specified using `PortfolioMAD`.

Properties

Setting Up the PortfolioMAD Object

`AssetList` — Names or symbols of assets in universe
`[]` (default) | cell array of character vectors | string array

Names or symbols of assets in the universe, specified as a cell array of character vectors or a string array.

Data Types: `cell` | `string`

`InitPort` — Initial portfolio
`[]` (default) | vector

Initial portfolio, specified as a vector.

Data Types: `double`

`Name` — Name for instance of PortfolioMAD object
`[]` (default) | character vector | string

Name for instance of the `PortfolioMAD` object, specified as a character vector.

Data Types: `char` | `string`

`NumAssets` — Number of assets in the universe

`[ ]` (default) | integer scalar

Number of assets in the universe, specified as an integer scalar.

Data Types: `double`

PortfolioMAD Object Constraints

`AEquality` — Linear equality constraint matrix

`[ ]` (default) | matrix

Linear equality constraint matrix, specified as a matrix. For more information, see “Linear Equality Constraints” on page 6-8.

Data Types: `double`

`AInequality` — Linear inequality constraint matrix

`[ ]` (default) | matrix

Linear inequality constraint matrix, specified as a matrix. For more information, see “Linear Inequality Constraints” on page 6-7.

Data Types: `double`

`bEquality` — Linear equality constraint vector

`[ ]` (default) | vector

Linear equality constraint vector, specified as a vector. For more information, see “Linear Equality Constraints” on page 6-8.

Data Types: `double`

`bInequality` — Linear inequality constraint

`[ ]` (default) | vector

Linear inequality constraint vector, specified as a vector. For more information, see “Linear Inequality Constraints” on page 6-7.

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. For more information, see “Group Constraints” on page 6-10.

Data Types: `double`

`GroupB` — Group B weights

`[ ]` (default) | matrix

Group B weights, specified as a matrix. For more information, see “Group Constraints” on page 6-10.

Data Types: `double`

`GroupMatrix` — Group membership matrix

[ ] (default) | matrix

Group membership matrix, specified as a matrix. For more information, see “Group Ratio Constraints” on page 6-10.

Data Types: `double`

`LowerBound` — Lower-bound constraint

[ ] (default) | vector

Lower-bound constraint, specified as a vector. For more information, see “'Simple' Bound Constraints” on page 6-8 and “'Conditional' Bound Constraints” on page 6-9.

Data Types: `double`

`LowerBudget` — Lower-bound budget constraint

[ ] (default) | scalar

Lower-bound budget constraint, specified as a scalar. For more information, see “Budget Constraints” on page 6-9.

Data Types: `double`

`LowerGroup` — Lower-bound group constraint

[ ] (default) | vector

Lower-bound group constraint, specified as a vector. For more information, see “Group Constraints” on page 6-10.

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. For more information, see “Group Ratio Constraints” on page 6-10.

Data Types: `double`

`UpperBound` — Upper-bound constraint

[ ] (default) | vector

Upper-bound constraint, specified as a vector. For more information, see “'Simple' Bound Constraints” on page 6-8 and “'Conditional' Bound Constraints” on page 6-9.

Data Types: `double`

`UpperBudget` — Upper-bound budget constraint

[ ] (default) | scalar

Upper-bound budget constraint, specified as a scalar. For more information, see “Budget Constraints” on page 6-9.

Data Types: `double`

`UpperGroup` — Upper-bound group constraint

[ ] (default) | vector

Upper-bound group constraint, specified as a vector. For more information, see “Group Constraints” on page 6-10.

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. For more information, see “Group Ratio Constraints” on page 6-10.

Data Types: `double`

`BoundType` — Type of bounds for each asset weight

`'Simple'` (default) | character vector with value `'Simple'` or `'Conditional'` | string with value `"Simple"` or `"Conditional"` | cell array of character vectors with values `'Simple'` or `'Conditional'` | string array with values `"Simple"` or `"Conditional"`

Type of bounds for each asset weight, specified as a scalar character vector or string, or a cell array of character vectors or a string array. For more information, see `setBounds`.

Data Types: `char` | `cell` | `string`

`MinNumAssets` — Minimum number of assets allocated in portfolio

`[]` (default) | numeric

Minimum number of assets allocated in portfolio, specified as a scalar numeric value. For more information, see `setMinMaxNumAssets` and “Cardinality Constraints” on page 6-12.

Data Types: `double`

`MaxNumAssets` — Maximum number of assets allocated in portfolio

`[]` (default) | numeric

Maximum number of assets allocated in portfolio, specified as a scalar numeric value. For more information, see `setMinMaxNumAssets` and “Cardinality Constraints” on page 6-12.

Data Types: `double`

`BuyTurnover` — Turnover constraint on purchases

`[]` (default) | scalar

Turnover constraint on purchases, specified as a scalar. For more information, see “Average Turnover Constraints” on page 6-11 and “One-way Turnover Constraints” on page 6-12.

Data Types: `double`

`SellTurnover` — Turnover constraint on sales

`[]` (default) | scalar

Turnover constraint on sales, specified as a scalar. For more information, see “Average Turnover Constraints” on page 6-11 and “One-way Turnover Constraints” on page 6-12.

Data Types: `double`

`Turnover` — Turnover constraint

`[]` (default) | scalar

Turnover constraint, specified as a scalar. For more information, see “Average Turnover Constraints” on page 6-11 and “One-way Turnover Constraints” on page 6-12.

Data Types: `double`

PortfolioMAD Object Modeling

`BuyCost` — Proportional purchase costs

`[]` (default) | vector

Proportional purchase costs, specified as a vector. For more information, see “Net Portfolio Returns” on page 6-3.

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. For more information, see “Net Portfolio Returns” on page 6-3.

Data Types: `double`

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 for portfolio
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 for portfolio
setMinMaxNumAssets	Set cardinality constraints on the number of assets invested in a portfolio
setBudget	Set up budget constraints for portfolio
setCosts	Set up proportional transaction costs for portfolio
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 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: []
    MinNumAssets: []
    MaxNumAssets: []
       BoundType: []
```

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` object directly to 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: []
     MinNumAssets: []
     MaxNumAssets: []
        BoundType: []
```

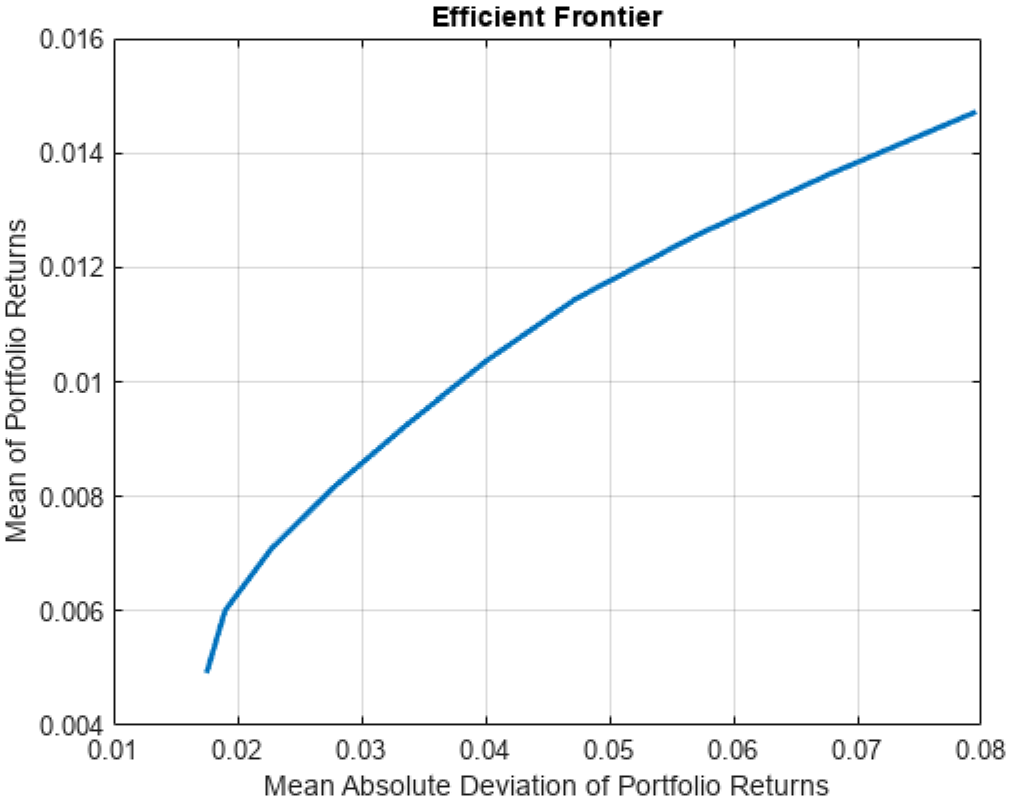
Note that the `LowerBound` property value undergoes scalar expansion since `AssetScenarios` provides the dimensions of the problem.

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 ];
```

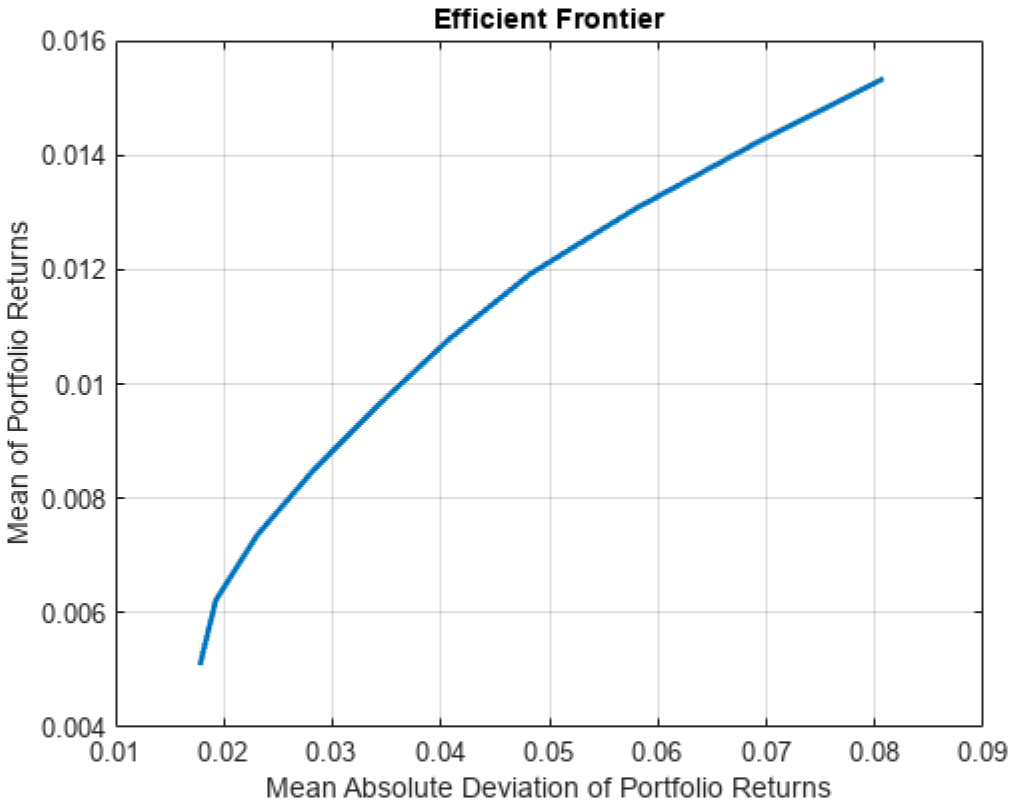
```
m = m/12;
C = C/12;

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 `PortfolioMAD` 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` object 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` object 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: []
     MinNumAssets: []
     MaxNumAssets: []
        BoundType: []
```

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);
```

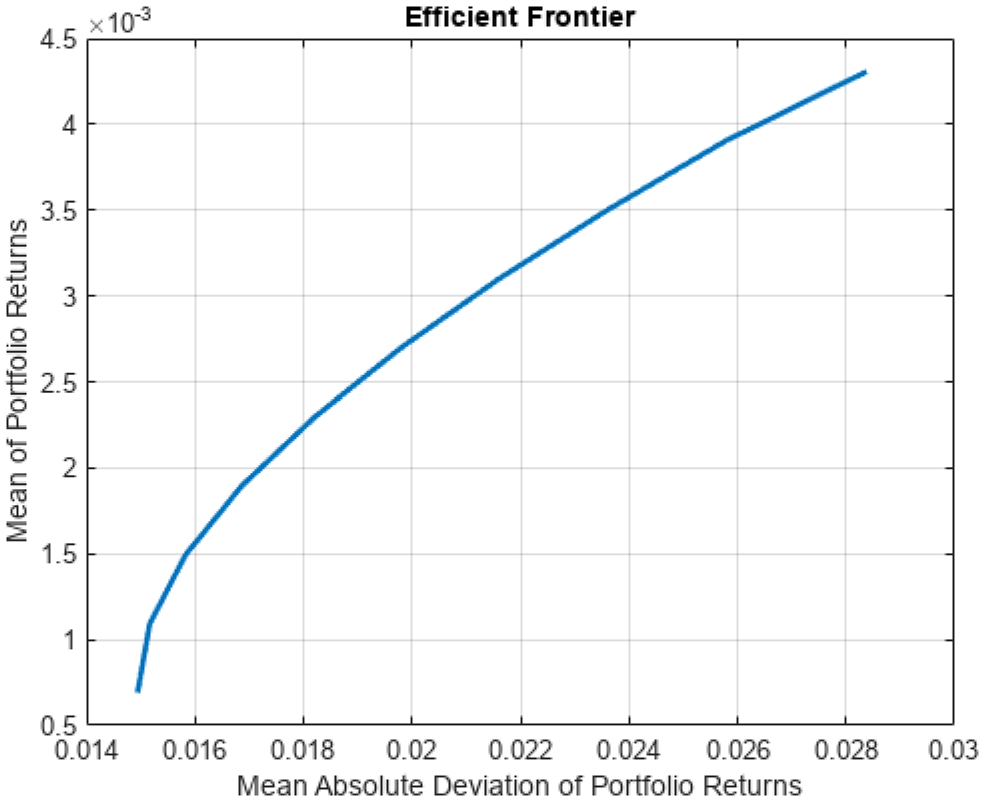
```
  PortfolioMAD with properties:

          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: []
     MinNumAssets: []
     MaxNumAssets: []
        BoundType: []
```

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    0.030785    0.074603     0.11383    0.13349    0
    AMZN           0           0           0          0    0
    CSCO           0           0           0          0    0
    DELL    0.010139           0           0          0    0
    EBAY           0           0           0          0    0
    GOOG      0.1607     0.35186     0.54435    0.74908    1
    HPQ     0.056834    0.024903           0          0    0
    IBM      0.45716     0.38008     0.29373    0.11743    0
    INTC           0           0           0          0    0
    MSFT     0.28438     0.16855    0.048097          0    0
    ORCL           0           0           0          0    0
    YHOO           0           0           0          0    0
```

More About

Mean-Absolute Deviation Portfolio Optimization

A MAD optimization problem is completely specified with three elements.

The three elements for a A MAD optimization problem are:

- 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 detailed information on the theoretical basis for mean-absolute deviation optimization, see “Portfolio Optimization Theory” on page 6-2.

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` object has shorter argument names that replace longer argument names associated with specific properties of the `PortfolioMAD` object.

For example, rather than enter `'AInequality'`, `PortfolioMAD` 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`

Version History

Introduced in R2013b

References

[1] For a complete list of references for the PortfolioMAD object, see “Portfolio Optimization” on page A-5.

See Also

`plotFrontier` | `estimateFrontier` | `setScenarios` | `PortfolioCVaR` | `Portfolio` | `nearcorr`

Topics

“Creating the PortfolioMAD Object” on page 6-21
“Common Operations on the PortfolioMAD Object” on page 6-28
“Working with MAD Portfolio Constraints Using Defaults” on page 6-48
“Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-34
“Validate the MAD Portfolio Problem” on page 6-76
“Estimate Efficient Portfolios Along the Entire Frontier for PortfolioMAD Object” on page 6-80
“Estimate Efficient Frontiers for PortfolioMAD Object” on page 6-97
“Postprocessing Results to Set Up Tradable Portfolios” on page 6-105
“Portfolio Optimization Theory” on page 6-2
“PortfolioMAD Object Workflow” on page 6-15
“PortfolioMAD Object Properties and Functions” on page 6-16
“Working with PortfolioMAD Objects” on page 6-16
“Setting and Getting Properties” on page 6-17
“Displaying PortfolioMAD Objects” on page 6-17
“Saving and Loading PortfolioMAD Objects” on page 6-17
“Estimating Efficient Portfolios and Frontiers” on page 6-17
“Arrays of PortfolioMAD Objects” on page 6-18
“Subclassing PortfolioMAD Objects” on page 6-19
“Conventions for Representation of Data” on page 6-19
“Choosing and Controlling the Solver for PortfolioMAD Optimizations” on page 6-91

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( ___ ,NumPorts,PortReturn)
portopt( ___ ,NumPorts,PortReturn)
```

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. By default, `portopt` returns 10 equally-spaced points on the efficient frontier.

`portopt` solves the "standard" mean-variance portfolio optimization problem for a long-only fully-invested investor with no additional constraints. Specifically, every portfolios on the efficient frontier has non-negative weights that sum to 1.

Note An alternative for portfolio optimization is to use the `Portfolio` object 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 4-17.

`[PortRisk,PortReturn,PortWts] = portopt( ___ ,NumPorts,PortReturn)` specifies options using one or more optional arguments in addition to the input arguments in the previous syntax.

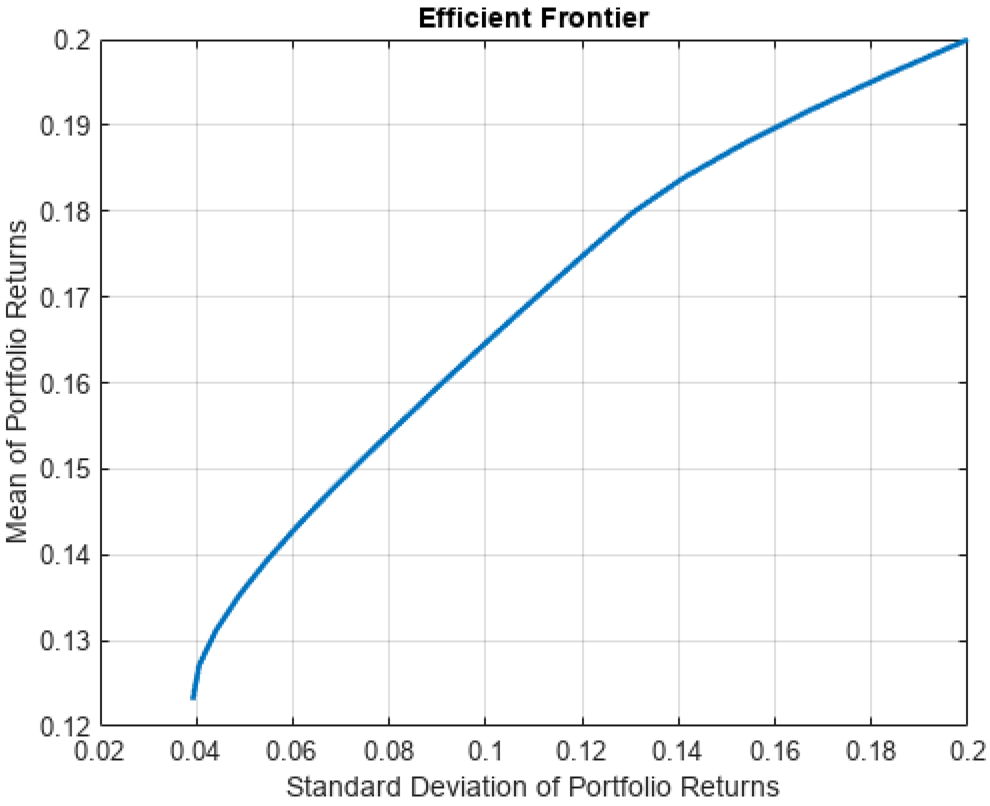
`portopt( ___ ,NumPorts,PortReturn)` returns a plot of the efficient frontier if `portopt` is invoked with no output arguments.

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)
```



Input Arguments

`ExpReturn` — Expected (mean) return of each asset

vector

Expected (mean) return of each asset, specified as a `1`-by-number of assets (`NASSETS`) vector.

Data Types: `double`

`ExpCovariance` — Covariance of the asset returns

matrix

Covariance of the asset returns, specified as a `NASSETS`-by-`NASSETS` matrix.

Data Types: `double`

`NumPorts` — Number of portfolios generated along the efficient frontier

`10` (default) | scalar numeric

(Optional) Number of portfolios generated along the efficient frontier, specified as a scalar numeric. Returns are equally spaced between the maximum possible return and the minimum risk point. If `NumPorts` is empty (entered as `[]`), `portopt` computes 10 equally spaced points. If you specify `1`, `portopt` returns the minimum-risk portfolio.

Note If not over-ridden by `PortReturn`, these portfolios are spaced evenly from the minimum to the maximum return on the efficient frontier. If `NumPorts = 1`, then the minimum-risk portfolio is computed (positive integer).

Data Types: `double`

`PortReturn` — Target portfolio returns to be computed on the efficient frontier
[ ] (default) | vector

(Optional) Target portfolio returns to be computed on the efficient frontier, specified as 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.

Note `portopt` requires that if you set `PortReturn`, `NumPorts` should be empty. If you specify `PortReturn` with a nonempty vector, `PortReturn` overrides `NumPorts`. If any returns in `PortReturn` fall outside the range of returns on the efficient frontier, `portopt` generates a warning and the efficient portfolios closest to the endpoints of the efficient frontier are computed.

Data Types: `double`

Output Arguments

`PortRisk` — Standard deviation of each portfolio
vector

Standard deviation of each portfolio, returned as a `NPORTS`-by-`1` vector.

`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.

`PortReturn` — expected return of each portfolio
vector

Expected return of each portfolio, returned as a `NPORTS`-by-`1` vector.

`PortWts` — Weights allocated to each asset
matrix

Weights allocated to each asset, returned as a `NPORTS`-by-`NASSETS` matrix. Each row represents a portfolio. The total of all weights in a portfolio is 1.

Version History

Introduced before R2006a

See Also

`ewstats` | `frontier` | `portstats` | `portcons` | `Portfolio`

Topics

"Portfolio Construction Examples" on page 3-5
"Plotting an Efficient Frontier Using portopt" on page 10-22
"Portfolio Selection and Risk Aversion" on page 3-7
"Bond Portfolio Optimization Using Portfolio Object" on page 10-30
"Active Returns and Tracking Error Efficient Frontier" on page 3-25
"portopt Migration to Portfolio Object" on page 3-11
"Analyzing Portfolios" on page 3-2
"Portfolio Optimization Functions" on page 3-3

portrand

Randomized portfolio risks, returns, and weights

Syntax

```
[PortRisk,PortReturn,PortWts] = portrand(Asset)
[PortRisk,PortReturn,PortWts] = portrand( ___ ,Return,Points,Method)
portrand(Asset,Return,Points,Method)
```

Description

`[PortRisk,PortReturn,PortWts] = portrand(Asset)` returns the risks, rates of return, and weights of random portfolio configurations. 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.

Note An alternative for portfolio optimization is to use the `Portfolio` object 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 4-17.

`[PortRisk,PortReturn,PortWts] = portrand( ___ ,Return,Points,Method)` returns the risks, rates of return, and weights of random portfolio configurations using optional arguments.

`portrand(Asset,Return,Points,Method)` plots the points representing each portfolio configuration. It does not return any data to the MATLAB workspace.

Input Arguments

`Asset` — Financial time series data

matrix

Financial time series data, specified as a matrix where each row is an observation and each column represents a single security.

Data Types: `double`

`Return` — Rate of return for corresponding security in `Asset`

`Return` computed by taking average value of each column of `Asset` (default) | row vector

(Optional) Rate of return for corresponding security in `Asset`, specified as a row vector,where each column represents the rate of return for the corresponding security.

Data Types: `double`

`Points` — Defines number of random points generated

`1000` (default) | numeric

(Optional) Defines number of random points generated, specified as a numeric.

Data Types: `double`

`Method` — Method to generate random portfolios from set of portfolios
`'uniform'` (default) | character vector with value `'uniform'` or `'geometric'`

(Optional) Method to generate random portfolios from set of portfolios, specified as a character vector for one of the following:

- `'uniform'` — Portfolio weights are generated that are uniformly distributed on the set of portfolio weights.
- `'geometric'` — Portfolio weights are generated 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.

Data Types: `double`

Output Arguments

`PortRisk` — Portfolio risk
vector

Portfolio risk, returned as a `POINTS`-by-`1` vector of standard deviations.

`PortReturn` — Portfolio return
vector

Portfolio return, returned as a `POINTS`-by-`1` vector of expected rates of return.

`PortWts` — Portfolio asset weights
matrix

Portfolio asset weights, returned as a `POINTS`-by-(number of securities) matrix of asset weights. Each row of `PortWts` is a different portfolio configuration.

Version History

Introduced before R2006a

References

[1] Bodie, Kane, and Marcus. *Investments.* Chapter 7.

See Also

`portror` | `portvar`

Topics
“Portfolio Construction Examples” on page 3-5

"Portfolio Optimization Functions" on page 3-3

portror

Portfolio expected rate of return

Syntax

```
R = portror(Return,Weight)
```

Description

`R = portror(Return,Weight)` returns a `1`-by-`M` vector for the expected rate of return.

Note An alternative for portfolio optimization is to use the `Portfolio` object 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 4-17.

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
```

Input Arguments

`Return` — Rates of return
matrix

Rates of return, specified as a `1`-by-`N` matrix. Each column of `Return` represents the rate of return for a single security.

Data Types: `double`

`Weight` — Weights
matrix

Weights, specified as a `M`-by-`N` matrix. Each row of `Weight` represents a different weighting combination of the assets in the portfolio.

Data Types: `double`

Output Arguments

R — Expected rate of return

vector

Expected rate of return, returned as a 1-by-M vector.

Version History

Introduced before R2006a

References

[1] Zvi Bodie, Alex Kane, Alan Marcus. *Investments.* McGraw-Hill Education; 10th edition (September 9, 2013).

See Also

portrand | portvar

Topics

"Portfolio Construction Examples" on page 3-5
"Portfolio Optimization Functions" on page 3-3

portsim

Monte Carlo simulation of correlated asset returns

Syntax

```
RetSeries = portsim(ExpReturn,ExpCovariance,NumObs)
RetSeries = portsim( ___ ,RetIntervals,NumSim,Method)
```

Description

`RetSeries = portsim(ExpReturn,ExpCovariance,NumObs)` 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.

Note An alternative for portfolio optimization is to use the `Portfolio` object 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 4-17.

`RetSeries = portsim( ___ ,RetIntervals,NumSim,Method)` specifies options using one or more optional arguments in addition to the input arguments in the previous syntax.

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 = 5×5
10^-3 ×
```

```
    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.

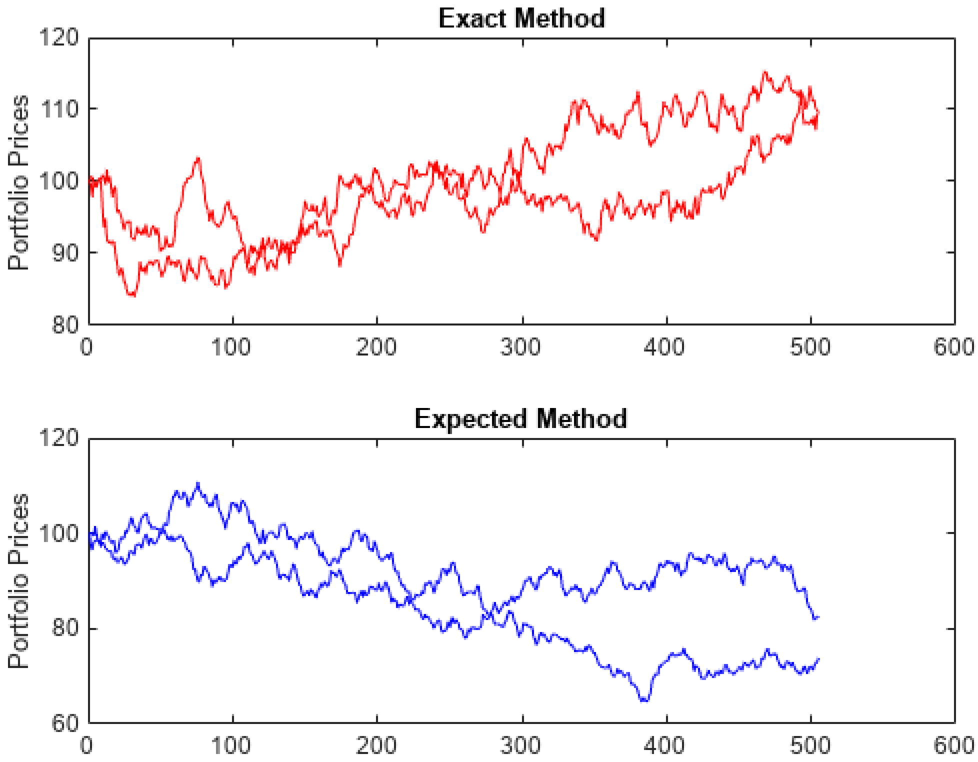
At this point, RetExact and RetExpected are both 504-by-5-by-2 arrays. Now assume an equally weighted portfolio formed from the five assets and create arrays of portfolio returns in which each column represents the portfolio return of the corresponding sample path of the simulated returns of the five 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 simulate a univariate geometric Brownian motion process. It is based on an example found in Hull, *Options, Futures, and Other Derivatives*, 5th Edition (see example 12.2 on page 236). In addition to verifying Hull's example, it also graphically illustrates the lognormal property of terminal stock prices by a rather large Monte Carlo simulation.

Assume 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;
```

`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.

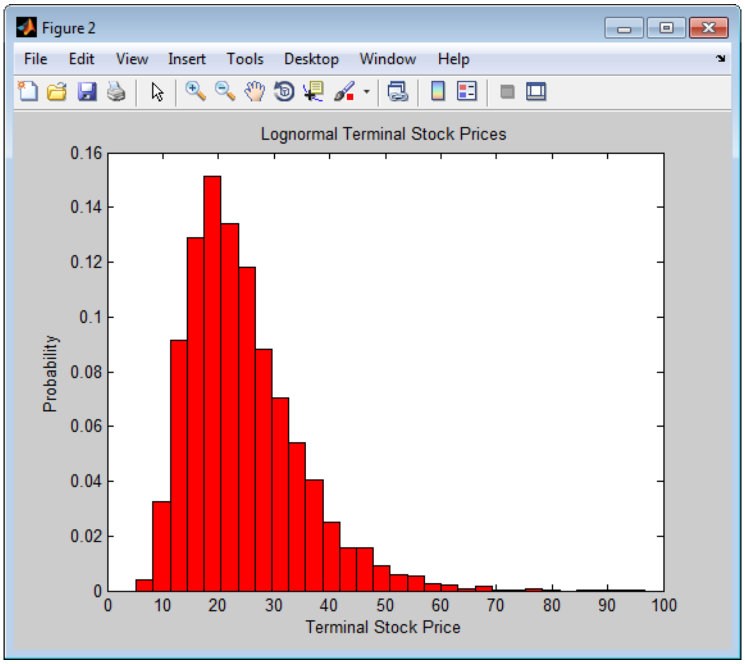
```
ExpValue = StartPrice*exp(ExpReturn)
ExpVar = ...
StartPrice*StartPrice*exp(2*ExpReturn)*(exp((ExpCovariance)) - 1)
```

```
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')
```



Input Arguments

ExpReturn — Expected (mean) return of each asset

vector

Expected (mean) return of each asset, specified as a 1-by-NASSETS vector.

Data Types: double

ExpCovariance — Asset return covariances

matrix

Asset return covariances, specified as an NASSETS-by-NASSETS matrix. ExpCovariance must be symmetric and positive semidefinite (no negative eigenvalues). The standard deviations of the returns are ExpSigma = sqrt(diag(ExpCovariance)). If ExpCovariance is not a symmetric positive semidefinite matrix, use nearcorr to create a positive semidefinite matrix for a correlation matrix.

Data Types: double

NumObs — Number of consecutive observations in the return time series

positive scalar integer

number of consecutive observations in the return time series, specified as a positive scalar integer. If NumObs is entered as the empty matrix [], the length of RetIntervals is used.

Data Types: double

RetIntervals — Interval times between observations

1 (default) | positive scalar | vector

(Optional) Interval times between observations, specified as a positive scalar or a number of observations NUMOBS-by-1 vector. If RetIntervals is not specified, all intervals are assumed to have length 1.

Data Types: double

NumSim — Number of simulated sample paths (realizations) of NUMOBS observations

1 (default) | positive scalar integer

(Optional) Number of simulated sample paths (realizations) of NUMOBS observations, specified as a positive scalar integer. The default value for NumSim is 1 (single realization of NUMOBS correlated asset returns).

Data Types: double

Method — Type of Monte Carlo simulation

'Exact' (default) | character vector

(Optional) Type of Monte Carlo simulation, specified as a character vector with one of the following values:

- '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 values 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.

Data Types: char

Output Arguments

RetSeries — Three-dimensional array of correlated, normally distributed, proportional asset returns

array

Three-dimensional array of correlated, normally distributed, proportional asset returns, returned as a NUMOBS-by-NASSETS-by-NUMSIM three-dimensional array.

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.

Version History

Introduced before R2006a

References

[1] Hull, J. C. *Options, Futures, and Other Derivatives.* Prentice-Hall, 2003.

See Also

ewstats | portopt | portstats | randn | ret2tick | squeeze | nearcorr

Topics

“Portfolio Construction Examples” on page 3-5
“Portfolio Optimization Functions” on page 3-3

portstats

Portfolio expected return and risk

Syntax

```
[PortRisk,PortReturn] = portstats(ExpReturn,ExpCovariance)
[PortRisk,PortReturn] = portstats( ___ ,Wts)
```

Description

`[PortRisk,PortReturn] = portstats(ExpReturn,ExpCovariance)` computes the expected rate of return and risk for a portfolio of assets.

Note An alternative for portfolio optimization is to use the `Portfolio` object 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 4-17.

`[PortRisk,PortReturn] = portstats( ___ ,Wts)` specifies options using one or more optional arguments in addition to the input arguments in the previous syntax.

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 = 2×1

    0.0560
    0.0550


PortReturn = 2×1

    0.1400
    0.1300
```

Input Arguments

ExpReturn — Expected (mean) return of each asset
vector

Expected (mean) return of each asset, specified as a 1-by-NASSETS vector.

Data Types: double

ExpCovariance — Asset return covariances
matrix

Asset return covariances, specified as an NASSETS-by-NASSETS matrix.

Data Types: double

Wts — Weights allocated to each asset
1/NASSETS (equally weighted) (default) | matrix

(Optional) Weights allocated to each asset, specified as an NPORTS-by-NASSETS matrix. Each row represents a different weighting combination of the assets in the portfolio. If Wts is not entered, weights of 1/NASSETS are assigned to each security.

Data Types: double

Output Arguments

PortRisk — Standard deviation of each portfolio
vector

Standard deviation of each portfolio, returned as an NPORTS-by-1 vector.

PortReturn — Expected return of each portfolio
vector

Expected return of each portfolio, returned an NPORTS-by-1 vector.

Version History

Introduced before R2006a

See Also

ewstats | portalloc

Topics

“Portfolio Construction Examples” on page 3-5
“Portfolio Selection and Risk Aversion” on page 3-7
“Active Returns and Tracking Error Efficient Frontier” on page 3-25
“Analyzing Portfolios” on page 3-2

“Portfolio Optimization Functions” on page 3-3

portvar

Variance for portfolio of assets

Syntax

```
V = portvar(Asset)
V = portvar(Asset,Weight)
```

Description

`V = portvar(Asset)` assigns each security an equal weight when calculating the portfolio variance.

Note An alternative for portfolio optimization is to use the `Portfolio` object 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 4-17.

`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`.

Examples

Compute Variance of Portfolio Assets

This example shows how to use `portvar` to compute the variance of portfolio assets. When you don't specify weights, `portvar` assigns each security an equal weight when calculating the portfolio variance.

```
load FundMarketCash
Returns = tick2ret(TestData);
Fund = Returns(:,1);
portvar(Fund)
```

```
ans = 5.3465e-04
```

Input Arguments

`Asset` — Asset returns

matrix

Asset Returns, specified as a `M`-by-`N` matrix of `M` asset returns for `N` securities.

Data Types: `double`

`Weight` — Portfolio weights

matrix

Portfolio weights, specified as a `R`-by-`N` matrix of `R` portfolio weights for `N` securities. Each row of `Weight` constitutes a portfolio of securities in `Asset`.

, specified as a scalar numeric value.

Data Types: `double`

Output Arguments

`V` — Variance of portfolio assets

numeric

Variance of portfolio assets, returned as a numeric value.

Version History

Introduced before R2006a

References

[1] Bodie, Kane, and Marcus. *Investments.* McGraw Hill, Chapter 7, 2013.

See Also

`portrand` | `portror`

Topics

"Portfolio Construction Examples" on page 3-5
"Portfolio Optimization Functions" on page 3-3

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. `portvrisk` calculates `ValueAtRisk` using a normal distribution.

Note An alternative for portfolio optimization is to use the `Portfolio` object 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 4-17.

`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 = 3×1

    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 = 2×1
10^7 ×

    3.6572
    1.8684
```

Compute the Maximum Potential Loss in the Value of a Portfolio Over One Period of Time Using Dollar Units for `PortReturn` and `PortRisk`

This example shows how to return the maximum potential loss in the value of a portfolio over one period of time, where the portfolio return (`PortReturn`) and risk (`PortRisk`) are specified in dollar units. The returned `ValueAtRisk` is also in dollar units. Note that the `PortValue` input for `portvrisk` is not used in this example. `PortValue` is only used as a scaling factor to convert from percent to dollars when `PortReturn` and `PortRisk` are specified on a percentage basis.

```
PortReturn = [2900000;1500000];
PortRisk = [30800000;15750000];
RiskThreshold = 0.10;
ValueAtRisk = portvrisk(PortReturn,PortRisk,RiskThreshold)
```

```
ValueAtRisk = 2×1
10^7 ×

    3.6572
    1.8684
```

Input Arguments

`PortReturn` — Expected return of each portfolio over period

scalar numeric | vector

Expected return of each portfolio over the period, specified as a scalar numeric or an `NPORTS`-by-`1` vector.

Data Types: `double`

`PortRisk` — Standard deviation of each portfolio over period

scalar numeric | vector

Standard deviation of each portfolio over period, specified as a scalar numeric or NPORTS-by-1 vector.

Data Types: double

RiskThreshold — Loss probability

0.05 (5%) (default) | scalar decimal | vector

(Optional) Loss probability, specified as a scalar decimal or an NPORTS-by-1 vector.

Data Types: double

PortValue — Total value of asset portfolio

1 (default) | scalar numeric | vector

(Optional) Total value of asset portfolio, specified as a scalar numeric or an NPORTS-by-1 vector.

Note The PortValue input is used as a scaling factor for ValueAtRisk. The ValueAtRisk output is computed using the PortReturn and PortRisk inputs first, and then scaled by PortValue. Therefore, if you specify PortValue in dollar units, then PortReturn and PortRisk must be given on a percentage basis. Conversely, when PortReturn and PortRisk are specified in dollar units, PortValue must be 1 (default value).

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.

Note When PortValue is the default value of 1, ValueAtRisk is presented as a percent. A value of 0 for ValueAtRisk indicates no losses.

Version History

Introduced before R2006a

See Also

portopt | Portfolio

Topics

"Portfolio Construction Examples" on page 3-5
"Portfolio Optimization Functions" on page 3-3

posvolidx

Positive volume index

Syntax

```
volume = posvolidx(Data)
volume = posvolidx( ___ ,Name,Value)
```

Description

`volume = posvolidx(Data)` calculates the positive volume index from the series of closing stock prices and trade volume.

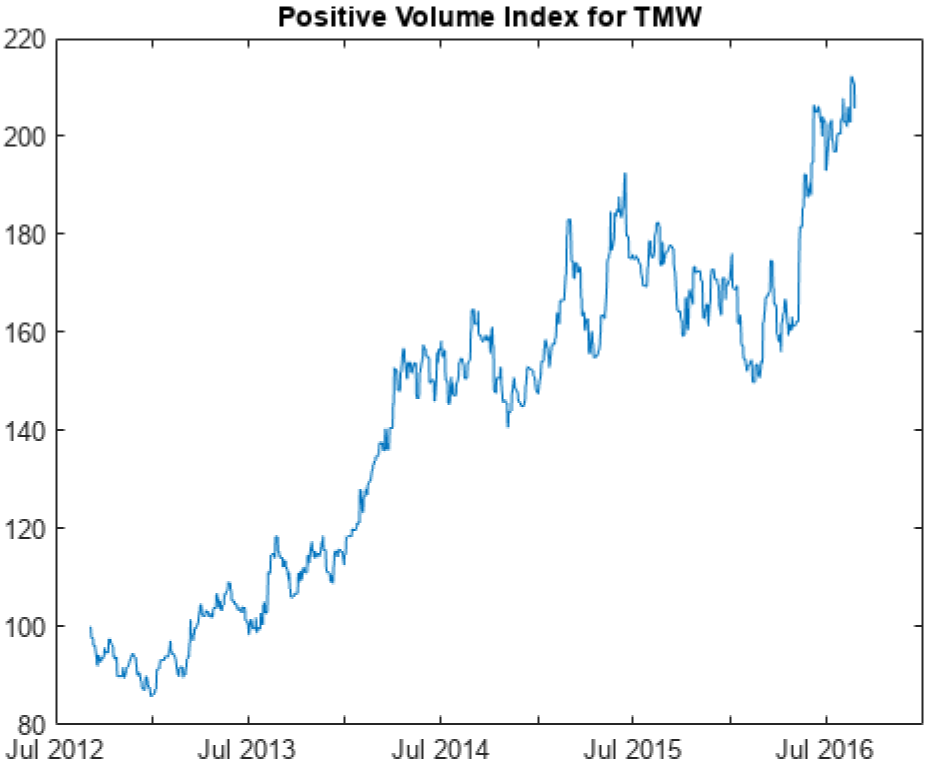
`volume = posvolidx( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Calculate the Positive Volume Index for a Data Series for a Stock

Load the file `SimulatedStock.mat`, which provides a timetable (`TMW`) for financial data for TMW stock.

```
load SimulatedStock.mat
volume = posvolidx(TMW);
plot(volume.Time,volume.PositiveVolume)
title('Positive Volume Index for TMW')
```



Input Arguments

`Data` — Data with closing prices and trade volume

matrix | table | timetable

Data with closing prices and trade volume, specified as a matrix, table, or timetable. For matrix input, `Data` is an `M`-by-2 with closing prices and trade volume stored in the first and second columns. Timetables and tables with `M` rows must contain variables named `'Close'` and `'Volume'` (case insensitive).

Data Types: `double` | `table` | `timetable`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `volume = posvolidx(TMW,'InitialValue',500)`

`InitialValue` — Initial value for positive volume index

`100` (default) | positive integer

Initial value for positive volume index, specified as the comma-separated pair consisting of `'InitialValue'` and a scalar positive integer.

Data Types: `double`

Output Arguments

`volume` — Positive volume index

matrix | table | timetable

Positive volume index, returned with the same number of rows (`M`) and the same type (matrix, table, or timetable) as the input `Data`.

More About

Positive Volume Index

Positive volume index shows the days when the trading volume of a particular security is substantially higher than other days.

Version History

Introduced before R2006a

R2023a: `fints` support removed for `Data` input argument

Behavior changed in R2023a

`fints` object support for the `Data` input argument is removed.

R2022b: Support for negative price data

Behavior changed in R2022b

The `Data` input accepts negative prices.

References

[1] Achelis, S. B. *Technical Analysis from A to Z.* Second Edition. McGraw-Hill, 1995, pp. 236–238.

See Also

`timetable` | `table` | `onbalvol` | `negvolidx`

Topics

"Use Timetables in Finance" on page 11-7
"Convert Financial Time Series Objects (fints) to Timetables" on page 11-2

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 = 11×1

    0.0616
    0.0609
    0.0658
    0.0590
    0.0647
    0.0655
    0.0606
    0.0601
    0.0642
    0.0621
      ⋮

ZeroDates = 11×1

      729907
      730364
      730439
      730500
      730667
      730668
      730971
      731032
      731033
      731321
      ⋮
```

Execute the function `prbyzero`.

```
BondPrices = prbyzero(Bonds, Settle, ZeroRates, ZeroDates)
```

```
BondPrices = 12×1

   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 = 12×1

   99.3750
   98.7980
  106.8270
   96.8750
  103.6249
  101.1250
  103.1250
```

```
   99.3637
  101.0000
  101.2500
      ⋮
```

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 a datetime array, string array, or date character vectors.
- `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 2-16.
- `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 a datetime array, string array, or date character vectors.
- 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 | char | string | datetime | table

Settle — Settlement date

datetime scalar | string scalar | date character vector

Settlement date, specified as a scalar datetime, string, or date character vector.

To support existing code, prbyzero also accepts serial date numbers as inputs, but they are not recommended.

Data Types: datetime | string | 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

ZeroDates — Observed dates for ZeroRates

datetime array | string array | date character vector

Observed dates for ZeroRates, specified as a NUMDATES-by-1 column vector using a datetime array, string array, or date character vectors.

To support existing code, prbyzero also accepts serial date numbers as inputs, but they are not recommended.

Data Types: datetime | string | char

Compounding — Compounding frequency of input ZeroRates when 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 the Financial Instruments Toolbox method getZeroRates for an IRDataCurve object with a Dates property to create a vector of dates and data acceptable for prbyzero. For more information, see “Converting an IRDataCurve or IRFunctionCurve Object” (Financial Instruments Toolbox).

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although prbyzero supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

tr2bonds | zbtprice | datetime

Topics

“Term Structure of Interest Rates” on page 2-29
“Fixed-Income Terminology” on page 2-15

prcroc

Price rate of change

Syntax

```
PriceChangeRate = prcroc(Data)
PriceChangeRate = prcroc( ___ ,Name,Value)
```

Description

`PriceChangeRate = prcroc(Data)` calculates the price rate-of-change, `PriceChangeRate`, from the series of closing stock prices. By default, the price rate-of-change is calculated between the current closing price and the closing price 12 periods ago.

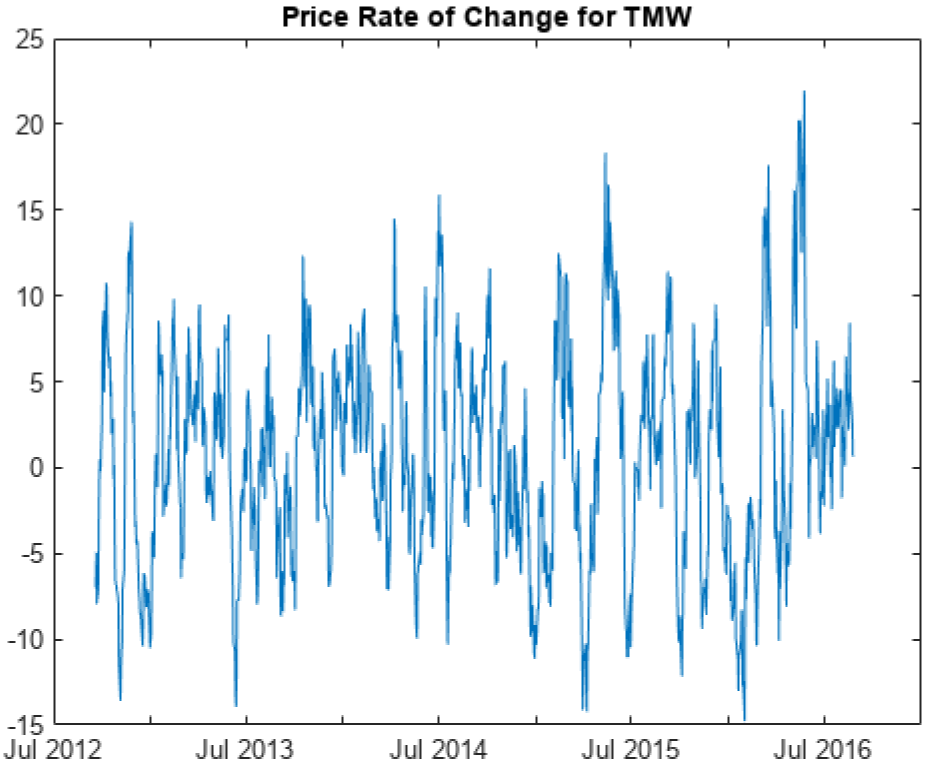
`PriceChangeRate = prcroc( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Calculate the Price Rate-of-Change for a Data Series for a Stock

Load the file `SimulatedStock.mat`, which provides a timetable (`TMW`) for financial data for TMW stock.

```
load SimulatedStock.mat
PriceChangeRate = prcroc(TMW);
plot(PriceChangeRate.Time,PriceChangeRate.PriceRoc)
title('Price Rate of Change for TMW')
```



Input Arguments

`Data` — Data for closing prices

matrix | table | timetable

Data for closing prices, specified as a matrix, table, or timetable. For matrix input, `Data` is an `M`-by-`1` matrix of closing prices. Timetables and tables with `M` rows must contain a variable named `'Close'` (case insensitive).

Data Types: `double` | `table` | `timetable`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `PriceChangeRate = prcroc(TMW,'NumPeriods',18)`

`NumPeriods` — Period difference

`12` (default) | positive integer

Period difference, specified as the comma-separated pair consisting of `'NumPeriods'` and a scalar positive integer.

Data Types: `double`

Output Arguments

`PriceChangeRate` — Closing price rate-of-change

matrix | table | timetable

Closing price rate-of-change, returned with the same number of rows (`M`) and the same type (matrix, table, or timetable) as the input `Data`.

Version History

Introduced before R2006a

R2023a: `fints` support removed for `Data` input argument

Behavior changed in R2023a

`fints` object support for the `Data` input argument is removed.

R2022b: Support for negative price data

Behavior changed in R2022b

The `Data` input accepts negative prices.

References

[1] Achelis, S. B. *Technical Analysis from A to Z.* Second Edition. McGraw-Hill, 1995, pp. 243–245.

See Also

`timetable` | `table` | `volroc`

Topics

“Use Timetables in Finance” on page 11-7
“Convert Financial Time Series Objects (fints) to Timetables” on page 11-2

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 = datetime(2000,10,14);
Maturity = datetime(2001,3,17);
Face = 100;
Discount = 0.087;
Basis = 2;

Price = prdisc(Settle,Maturity,Face, Discount, Basis)
```

```
Price = 96.2783
```

Input Arguments

`Settle` — Settlement date

datetime scalar | string scalar | date character vector

Settlement date, specified as a scalar datetime, string, or date character vector.

Settle must be earlier than `Maturity`.

To support existing code, `prdisc` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `datetime` | `string` | `char`

`Maturity` — Maturity date

datetime scalar | string scalar | date character vector

Maturity date, specified as a scalar datetime, string, or date character vector.

To support existing code, `prdisc` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `datetime` | `string` | `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 2-16.

Data Types: `double`

Output Arguments

`Price` — Price of discounted security

numeric

Price of discounted security, returned as a numeric value.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although `prdisc` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)

y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Mayle. *“Standard Securities Calculation Methods.”* Volumes I-II, 3rd edition. Formula 2.

See Also

`acrudisc` | `bndprice` | `discrate` | `prmat` | `ylddisc` | `datetime`

Topics

“Pricing Functions” on page 2-21
“Fixed-Income Terminology” on page 2-15

priceandvol

Price and Volume chart

Note `priceandvol` is updated to accept data input as a matrix, `timetable`, or `table`.

The syntax for `priceandvol` has changed. Previously, when using table input, the first column of dates could be a datetime array, date character vectors, or serial date numbers, and you were required to have specific number of columns.

When using table input, the new syntax for `priceandvol` supports:

- No need for time information. If you want to pass in date information, use timetable input.
- No requirement of specific number of columns. However, you must provide valid column names. `linebreak` must contain columns named `'open'`, `'high'`, `'low'`, `'close'`, and `'volume'` (case insensitive).

Syntax

```
priceandvol(Data)
h = priceandvol(Data)
```

Description

`priceandvol(Data)` plots two charts from a series of opening, high, low, closing prices, and traded volume. Opening, high, low, and closing prices are on one axis and the volume series are on a second axis.

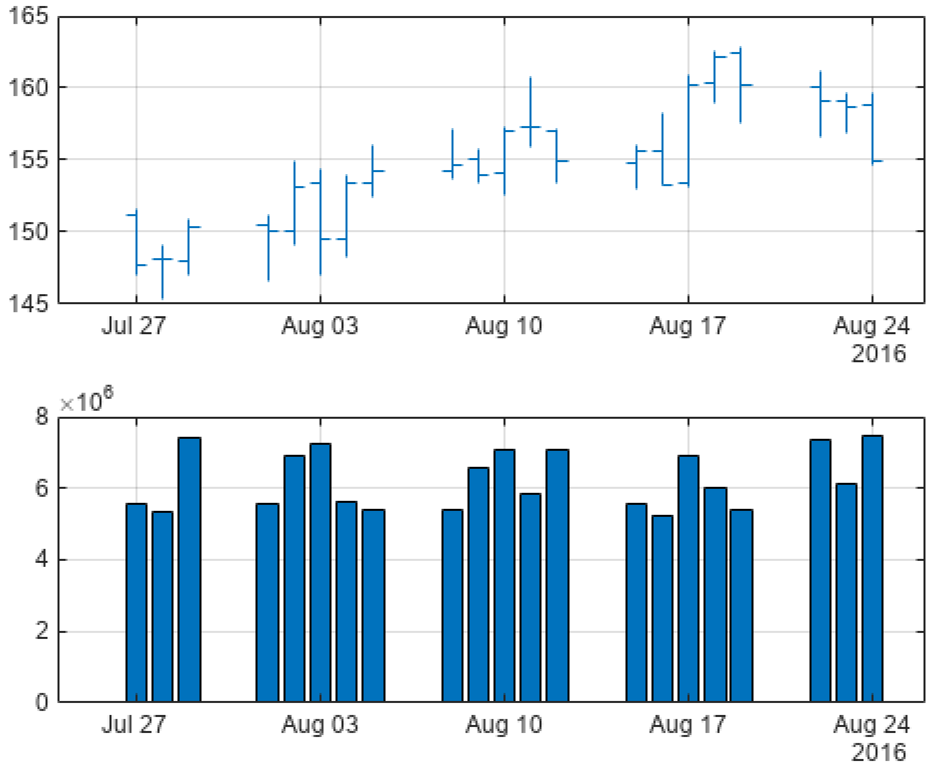
`h = priceandvol(Data)` adds a Graphic handle for the figure.

Examples

Generate a Price and Volume Chart for a Data Series for a Stock

Load the file `SimulatedStock.mat`, which provides a timetable (`TMW`) for financial data for TMW stock. This price and volume chart for the TMW stock contains the open, high, low, close, and volume for the most recent 21 days.

```
load SimulatedStock.mat
priceandvol(TMW(end-20:end,:));
```



Input Arguments

`Data` — Data for opening, high, low, closing prices, and volume traded

matrix | table | timetable

Data for opening, high, low, closing prices, and volume traded, specified as a matrix, table, or timetable. For matrix input, `Data` is an `M`-by-`5` matrix of opening, high, low, closing prices and traded volume. Timetables and tables with `M` rows must contain variables named `'Open'`, `'High'`, `'Low'`, `'Close'`, and `'Volume'` (case insensitive).

Data Types: `double` | `table` | `timetable`

Output Arguments

`h` — Graphic handle of the figure

handle object

Graphic handle of the figure, returned as a handle object.

Version History

Introduced in R2008a

R2022b: Support for negative price data

Behavior changed in R2022b

The `Data` input accepts negative prices.

See Also

`timetable` | `table` | `movavg` | `linebreak` | `highlow` | `kagi` | `volarea` | `candle` | `pointfig`

Topics

"Use Timetables in Finance" on page 11-7
"Convert Financial Time Series Objects (fints) to Timetables" on page 11-2

prmat

Price with interest at maturity

Syntax

```
[Price,AccruInterest] = prmat(Settle,Maturity,Issue,Face,CouponRateYield)
[Price,AccruInterest] = prmat( ___ ,Basis)
```

Description

`[Price,AccruInterest] = prmat(Settle,Maturity,Issue,Face,CouponRateYield)` 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

Compute Price and Accrued Interest of a Security That Pays Interest at Maturity

This example shows how to compute the price and accrued interest of a security that pays interest at maturity.

```
Settle = '02/07/2002';
Maturity = '04/13/2002';
Issue = '10/11/2001';
Face = 100;
CouponRate = 0.0608;
Yield = 0.0608;
Basis = 1;

[Price, AccruInterest] = prmat(Settle, Maturity, Issue, Face,...
CouponRate, Yield, Basis)
```

```
Price = 99.9784

AccruInterest = 1.9591
```

Compute Price and Accrued Interest of a Security That Pays Interest at Maturity Using datetime Inputs

This example shows how to use `datetime` inputs compute the price and accrued interest of a security that pays interest at maturity.

```
Settle = datetime(2002,2,7);
Maturity = datetime(2002,4,13);
Issue = datetime(2001,10,11);
Face = 100;
```

```
CouponRate = 0.0608;
Yield = 0.0608;
Basis = 1;

[Price, AccruInterest] = prmat(Settle, Maturity, Issue, Face, CouponRate, Yield, Basis)
```

```
Price = 99.9784

AccruInterest = 1.9591
```

Input Arguments

`Settle` — Settlement date of security

datetime scalar | string scalar | date character vector

Settlement date of the security, specified as a scalar datetime, string, or date character vector. The `Settle` date must be before the `Maturity` date.

To support existing code, `prmat` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Maturity` — Maturity date of security

datetime scalar | string scalar | date character vector

Maturity date of the security, specified as a scalar datetime, string, or date character vector.

To support existing code, `prmat` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `datetime` | `string`

`Issue` — Issue date

datetime scalar | string scalar | date character vector

Issue date of the security, specified as a scalar datetime, string, or date character vector.

To support existing code, `prmat` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `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 2-16.

Data Types: `double`

Output Arguments

`Price` — Security price

numeric

Security price, returned as a numeric value.

`AccruInterest` — Accrued interest

numeric

Accrued interest for security, returned as a numeric value.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although prmat supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

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

See Also

acrudisc | bndprice | bndyield | ylddisc | yldtbill | datetime

Topics

“Yield Functions” on page 2-22
“Yield Conventions” on page 2-21

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(2002,2,10), datetime(2002,8,6), 1000, 0.0377)
```

```
Price = 981.4642
```

Input Arguments

`Settle` — Settlement date for Treasury bill

datetime scalar | string scalar | date character vector

Settlement date for the Treasury bill, specified as a scalar datetime, string, or date character vector. The `Settle` date must be before the `Maturity` date.

To support existing code, `prtbill` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Maturity` — Maturity date for Treasury bill

datetime scalar | string scalar | date character vector

Maturity date for the Treasury bill, specified as a scalar datetime, string, or date character vector.

To support existing code, `prtbill` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `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.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although `prtbill` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Bodie, Kane, and Marcus. *Investments.* McGraw-Hill Education, 2013.

See Also

`beytbill` | `yldtbill` | `datetime`

Topics

“Computing Treasury Bill Price and Yield” on page 2-26

“Treasury Bills Defined” on page 2-25

pvfix

Present value with fixed periodic payments

Syntax

```
PresentVal = pvfix(Rate,NumPeriods,Payment)
PresentVal = pvfix( ___ ,ExtraPayment,Due)
```

Description

`PresentVal = pvfix(Rate,NumPeriods,Payment)` computes the present value of a series of equal payments.

`PresentVal = pvfix( ___ ,ExtraPayment,Due)` adds optional arguments.

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
```

Input Arguments

`Rate` — Periodic interest rate
decimal

Periodic interest rate, specified as a decimal.

Data Types: `double`

`NumPeriods` — Number of periods
integer

Number of periods, specified as an integer.

Data Types: `double`

`Payment` — Periodic payment
numeric

Periodic payment., specified as a numeric.

Data Types: `double`

`ExtraPayment` — Payment received other than `Payment` in the last period

`0` (default) | numeric

(Optional) Payment received other than `Payment` in the last period, specified as a numeric.

Data Types: `double`

`Due` — Indicator for when payments are due

`0` (default) | logical with value of `1` or `0`

(Optional) Indicator for when payments are due, specified as a logical with a value of `0` = end of period (default), or `1` = beginning of period.

Data Types: `logical`

Output Arguments

`PresentVal` — Present value

numeric

Present value, returned as a series of equal payments.

Version History

Introduced before R2006a

See Also

`fvfix` | `fvvar` | `payper` | `pvvar`

Topics

"Analyzing and Computing Cash Flows" on page 2-11

pvtrend

Price and Volume Trend (PVT)

Syntax

```
trend = pvtrend(Data)
```

Description

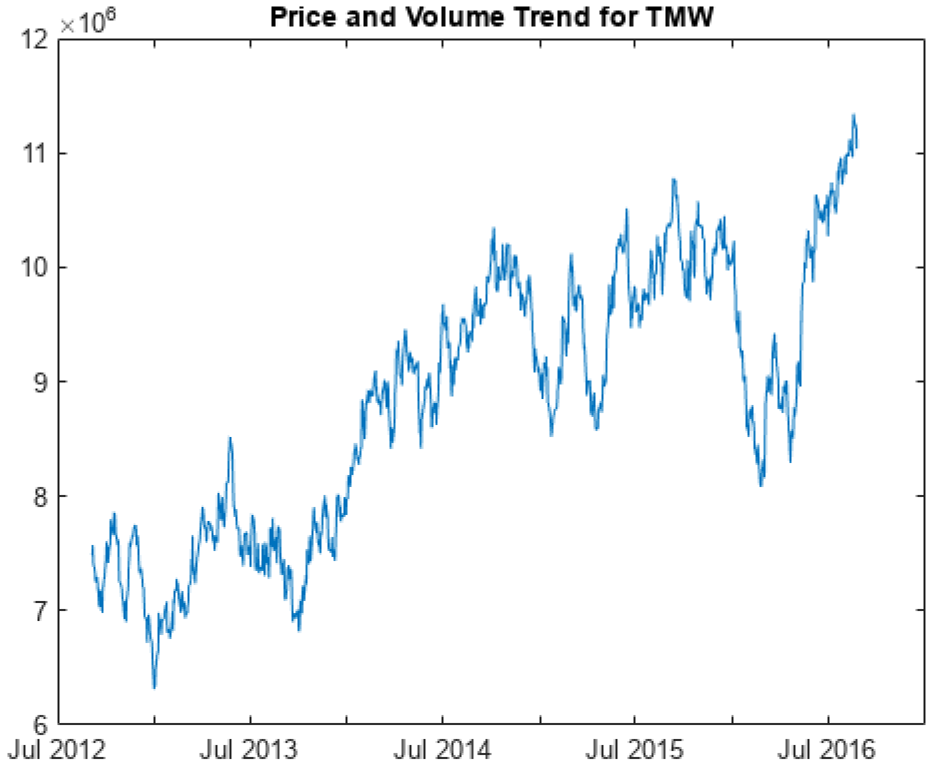
`trend = pvtrend(Data)` calculates the Price and Volume Trend (PVT) from the series of closing stock prices and trade volume.

Examples

Calculate the Price and Volume Trend for a Data Series for a Stock

Load the file `SimulatedStock.mat`, which provides a timetable (`TMW`) for financial data for TMW stock.

```
load SimulatedStock.mat
trend = pvtrend(TMW);
plot(trend.Time,trend.PriceVolumeTrend)
title('Price and Volume Trend for TMW')
```



Input Arguments

`Data` — Data for closing prices and trade volume

matrix | table | timetable

Data for closing prices and trade volume, specified as a matrix, table, or timetable. For matrix input, `Data` is an `M`-by-2 matrix of closing prices and trade volume. Timetables and tables with `M` rows must contain variables named `'Close'` and `'Volume'` (case insensitive).

Data Types: `double` | `table` | `timetable`

Output Arguments

`trend` — Price and volume trend

matrix | table | timetable

Price and volume trend (PVT), returned with the same number of rows (`M`) and the same type (matrix, table, or timetable) as the input `Data`.

Version History

Introduced before R2006a

R2023a: `fints` support removed for `Data` input argument

Behavior changed in R2023a

`fints` object support for the `Data` input argument is removed.

R2022b: Support for negative price data

Behavior changed in R2022b

The `Data` input accepts negative prices.

References

[1] Achelis, S. B. *Technical Analysis from A to Z.* Second Edition. McGraw-Hill, 1995, pp. 239–240.

See Also

`timetable` | `table` | `onbalvol` | `volroc` | `tsaccel`

Topics

"Use Timetables in Finance" on page 11-7
"Convert Financial Time Series Objects (fints) to Timetables" on page 11-2

pvvar

Present value of varying cash flow

Syntax

```
PresentVal = pvvar(CashFlow,Rate)
PresentVal = pvvar( ___ ,CFDates)
```

Description

`PresentVal = pvvar(CashFlow,Rate)` calculates present value of a varying cash flow.

`PresentVal = pvvar( ___ ,CFDates)` adds an optional argument for `CFDates`.

Examples

Calculate Present Value for Regular and Irregular Cash Flow

Calculate the net present value for a regular and irregular cash flow.

Regular Cash Flow

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)
```

```
PresentVal = 1.7154e+03
```

Irregular Cash Flow

An investment of $10,000 returns this irregular cash flow. The original investment and its date are included. The periodic interest rate is 9%.

January 12, 1987 - ($10000)

February 14, 1988 - $1500

March 3, 1988 - $2000

June 14, 1988 - $3000

December 1, 1988 - $4000

To calculate the net present value of this irregular cash flow:

```
CashFlow = [-10000, 1500, 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)
```

```
PresentVal = -768.1461
```

The net present value of the same investment under different discount rates of 7%, 9%, and 11% is obtained by:

```
PresentVal = pvvar(repmat(CashFlow,3,1)', [.07 .09 .11], CFDates)
```

```
PresentVal = 1×3
10^3 ×

   -0.5099   -0.7681   -1.0146
```

Input Arguments

CashFlow — Cash flow amounts

vector

Cash flow amounts, specified as 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.

Data Types: double

Rate — Periodic interest rate

decimal

Periodic interest rate, specified as a decimal. 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.

Data Types: double

CFDates — Indicates irregular cash flow

datetime array | string array | cell array of date character vectors

(Optional) Indicates irregular cash flow, specified as a datetime array, string array, or cell array of date character vectors 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.

To support existing code, pvvar also accepts serial date numbers as inputs, but they are not recommended.

Data Types: cell | string | char | datetime

Output Arguments

PresentVal — Present value
numeric

Present value, returns the net present value of a varying cash flow. Present value is calculated at the time the first cash flow occurs.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended
Not recommended starting in R2022b

Although pvvar supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

fvfix | fvvar | irr | payuni | pvfix | datetime

Topics
“Analyzing and Computing Cash Flows” on page 2-11

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 = 8×1

    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 = 8×1

    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 = datetime(2013,2,1);

CurveDates = [datetime(2014,2,1)
    datetime(2015,2,1)
    datetime(2016,2,1)
    datetime(2018,2,1)
    datetime(2020,2,1)
    datetime(2023,2,1)
    datetime(2033,2,1)
    datetime(2043,2,1)];

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 Dates] = pyld2zero(OriginalParRates, CurveDates, Settle, ...
'OutputCompounding', OutputCompounding, 'OutputBasis', OutputBasis, ...
'InputCompounding', InputCompounding, 'InputBasis', InputBasis)
```

```
ZeroRates = 8×1

    0.0011
    0.0030
    0.0064
    0.0144
    0.0207
    0.0261
    0.0329
    0.0356


Dates = 8x1 datetime
   01-Feb-2014
   01-Feb-2015
   01-Feb-2016
   01-Feb-2018
   01-Feb-2020
   01-Feb-2023
   01-Feb-2033
   01-Feb-2043
```

Demonstrate a Roundtrip From pyld2zero to zero2pyld

Given the following a par yield curve and its maturity dates, return the `ZeroRates`.

```
Settle = datetime(2013,2,1);

CurveDates = [datetime(2014,2,1)
    datetime(2015,2,1)
    datetime(2016,2,1)
    datetime(2018,2,1)
    datetime(2020,2,1)
    datetime(2023,2,1)
    datetime(2033,2,1)
    datetime(2043,2,1)];

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 = 8×1

    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 = 8×1

    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
```

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

datetime array | string array | date character vector

Maturity dates which correspond to the input ParRates, specified as a NUMBONDS-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, pyld2zero also accepts serial date numbers as inputs, but they are not recommended.

Data Types: datetime | string | char

Settle — Common settlement date for ZeroRates

datetime scalar | string scalar | date character vector

Common settlement date for input ParRates, specified as a scalar datetime, string, or data character vector.

To support existing code, pyld2zero also accepts serial date numbers as inputs, but they are not recommended.

Data Types: datetime | string | char

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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 2-16.

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 2-16.

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`

datetime | serial date number

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. Use the function `datestr` to convert serial date numbers to formatted date character vectors.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended
Not recommended starting in R2022b

Although `pyld2zero` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`fwd2zero` | `zero2fwd` | `getForwardRates` | `datetime` | `disc2zero` | `zero2disc` | `zero2pyld` | `zbtprice` | `zbtyield` | `datetime`

Topics

"Term Structure of Interest Rates" on page 2-29
"Sensitivity of Bond Prices to Interest Rates" on page 10-2
"Bond Prices and Yield Curve Parallel Shifts" on page 10-9
"Bond Prices and Yield Curve Nonparallel Shifts" on page 10-12
"Term Structure Analysis and Interest-Rate Swaps" on page 10-18
"Fixed-Income Terminology" on page 2-15

renko

Renko

Note renko is updated to accept data input as a matrix, timetable, or table.

The syntax for renko has changed. Previously, when using table input, the first column of dates could be a datetime array, date character vectors, or serial date numbers, and you were required to have specific number of columns.

When using table input, the new syntax for renko supports:

- No need for time information. If you want to pass in date information, use timetable input.
- No requirement of specific number of columns. However, you must provide valid column names. renko must contain a column named ‘price’ (case insensitive).

Syntax

```
renko(Data)
renko(Data,Threshold,)
h = renko(ax,___)
```

Description

renko(Data) plots a Renko chart from a series of prices of a security.

renko(Data,Threshold,) adds an optional argument for Threshold.

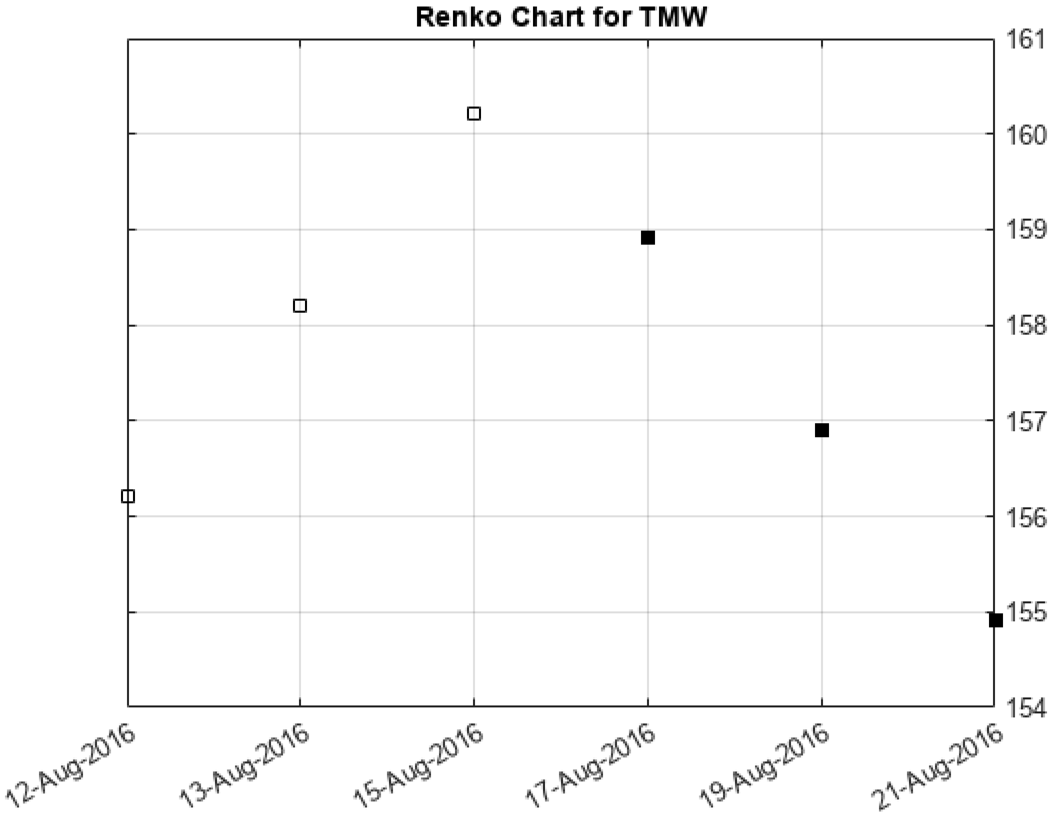
h = renko(ax,___) adds an optional argument for ax.

Examples

Generate a Renko Chart for a Data Series for a Stock

Load the file SimulatedStock.mat, which provides a timetable (TMW) for financial data for TMW stock. This example shows how to plot a Renko chart for the most recent 21 days. Note that the variable name of asset price is be renamed to 'Price' (case insensitive).

```
load SimulatedStock.mat
TMW.Properties.VariableNames{'Close'} = 'Price';
renko(TMW(end-8:end,:),2)
title('Renko Chart for TMW')
```



Input Arguments

`Data` — Data for a series of prices

matrix | table | timetable

Data for a series of prices, specified as a matrix, table, or timetable. Timetables and tables with `M` rows must contain a variable named `'Price'` (case insensitive).

Data Types: `double` | `table` | `timetable`

`Threshold` — (Optional) Least price change value when adding a new box

`1` (default) | positive numeric

Least price change value when adding a new box, specified as a scalar positive numeric value.

Data Types: `double`

`ax` — Valid axis object

current axes (`ax = gca`) (default) | axes object

(Optional) Valid axis object, specified as an axes object. The renko plot is created in the axes specified by `ax` instead of in the current axes (`ax = gca`). The option `ax` can precede any of the input argument combinations.

Data Types: `object`

Output Arguments

h — Graphic handle of the figure
handle object

Graphic handle of the figure, returned as a handle object.

Version History

Introduced in R2008a

R2022b: Support for negative price data
Behavior changed in R2022b

The `Data` input accepts negative prices.

See Also

`timetable` | `table` | `movavg` | `linebreak` | `highlow` | `kagi` | `priceandvol` | `volarea` | `candle` | `pointfig`

Topics
“Use Timetables in Finance” on page 11-7
“Convert Financial Time Series Objects (fints) to Timetables” on page 11-2

ret2tick

Convert return series to price series

Syntax

```
[TickSeries,TickTimes] = ret2tick(Data)
[TickSeries,TickTimes] = ret2tick( ___ ,Name,Value)
```

Description

`[TickSeries,TickTimes] = ret2tick(Data)` computes prices from the start prices of `NASSET` assets and `NUMOBS` return observations.

`[TickSeries,TickTimes] = ret2tick( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Convert a Return Series to a Price Series

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,'ReturnIntervals',RetIntervals,...
    'StartTime',StartTime)
```

```
TickSeries = 4×2

    1.0000    1.0000
    1.1000    1.1200
    1.1550    1.1648
    1.0973    1.2230


TickTimes = 4x1 datetime
   18-Dec-2000
   18-Jun-2001
   17-Sep-2001
   18-Dec-2001
```

Convert a Price Series to a Return Series Using `timetable` Input

Use `timetable` input to convert a price series to a return series, given periodic returns of two stocks observed in the first, second, third, and fourth quarters.

```
RetSeries = [0.10 0.12
             0.05 0.04
            -0.05 0.05];

RetTimes = datetime({'6/18/2001','9/17/2001','12/18/2001'},'InputFormat','MM/dd/uuuu','Locale','e
RetSeries = array2timetable(RetSeries,'RowTimes',RetTimes);
StartTime = datetime('12/18/2000','InputFormat','MM/dd/uuuu','Locale','en_US');


[TickSeries,TickTimes] = ret2tick(RetSeries,'StartTime',StartTime)
```

```
TickSeries=4×2 timetable
       Time        RetSeries1    RetSeries2
    ___________    __________    __________

    18-Dec-2000          1             1   
    18-Jun-2001        1.1          1.12   
    17-Sep-2001      1.155        1.1648   
    18-Dec-2001     1.0973         1.223   


TickTimes = 4x1 datetime
   18-Dec-2000
   18-Jun-2001
   17-Sep-2001
   18-Dec-2001
```

Input Arguments

`Data` — Data for asset returns

matrix | table | timetable

Data for asset returns, specified as a `NUMOBSNASSETS` matrix, `table`, or `timetable`. The returns are not normalized by the time increments between successive price observations.

Data Types: `double` | `table` | `timetable`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[TickSeries,TickTimes] = ret2tick(RetSeries,'StartTime',StartTime)`

`StartPrice` — Initial prices for each asset

`1` for all assets (default) | numeric

Initial prices for each asset, specified as the comma-separated pair consisting of `'StartPrice'` and a `NASSETS`-by-`1` vector indicating initial prices for each asset, or a single scalar initial price applied to all assets.

Data Types: `double`

`ReturnIntervals` — Return interval between prices
`1` for all assets (default) | numeric | duration | calendar duration

Return interval between prices, specified as the comma-separated pair consisting of `'ReturnIntervals'` and a scalar return interval applied to all returns, or a vector of length `NUMOBS` return intervals between successive returns. `ReturnIntervals` is defined as:

```
ReturnIntervals(t) = TickTimes(t) - TickTimes(t-1).
```

Note If the type of `Data` is a timetable, `ReturnIntervals` is ignored.

Data Types: `double`

`StartTime` — Starting time for first observation applied to the prices of all assets
`0` if `ReturnIntervals` is numeric (default) | numeric | duration | calendar duration

Starting time for first observation applied to the price series of all assets, specified as the comma-separated pair consisting of `'StartTime'` and a scalar string, character vector, double, or datetime.

Note If `ReturnIntervals` is a duration or calendar duration value, the default for `StartTime` is `datetime('today')`.

If `Data` is a timetable and `StartTime` is not specified, the resulting asset prices in the first period are not reported.

Data Types: `double` | `string` | `char` | `datetime`

`Method` — Method to convert asset returns to prices
`'Simple'` (default) | character vector with value of `'Simple'` or `'Continuous'` | string with value of `"Simple"` or `"Continuous"`

Method to convert asset returns to prices, specified as the comma-separated pair consisting of `'Method'` and a string or character vector indicating the method to convert asset prices to returns.

If the method is `'Simple'`, then simple periodic returns are used:

```
TTickSeries(t) = TickSeries(t-1)*(1 + ReturnSeries(t)).
```

If the method is `'Continuous'`, then continuous returns are used:

```
TickSeries(t) = TickSeries(t-1)*exp(ReturnSeries(t)).
```

Data Types: `char` | `string`

Output Arguments

`TickSeries` — Time series array of asset prices

matrix | table | timetable

Time series array of asset prices, returned as `NUMBOBS+1`-by-`NASSETS` time series of asset prices of the same type (matrix, table, or timetable) as the input `Data`. The first row contains the oldest prices and the last row contains the most recent. Prices across a given row are assumed to occur at the same time for all columns, and each column is a price series of an individual asset.

`TickTimes` — Observation times associated with prices in `TickSeries`

vector

Observation times associated with the prices in `TickSeries`, returned as a `NUMBOBS+1` length column vector of monotonically increasing observation times associated with the prices in `TickSeries`. The initial time is `StartTime`. For matrix and table `Data`, sequential observations occur at increments specified in `ReturnIntervals` and for `Data` timetables, sequential observations are derived from times and dates in `Data`.

Version History

Introduced before R2006a

R2022b: Support for negative price data

Behavior changed in R2022b

The `Data` input accepts negative prices.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports input `Data` that is specified as a tall column vector, a tall table, or a tall timetable. For more information, see `tall` and “Tall Arrays”.

See Also

`tick2ret` | `datetime` | `timetable` | `table`

Topics

“Use Timetables in Finance” on page 11-7
“Returns with Negative Prices” on page 2-32

rsindex

Relative Strength Index (RSI)

Syntax

```
index = rsindex(Data)
index = rsindex( ___ ,Name,Value)
```

Description

`index = rsindex(Data)` calculates the Relative Strength Index (RSI) from the series of closing stock prices.

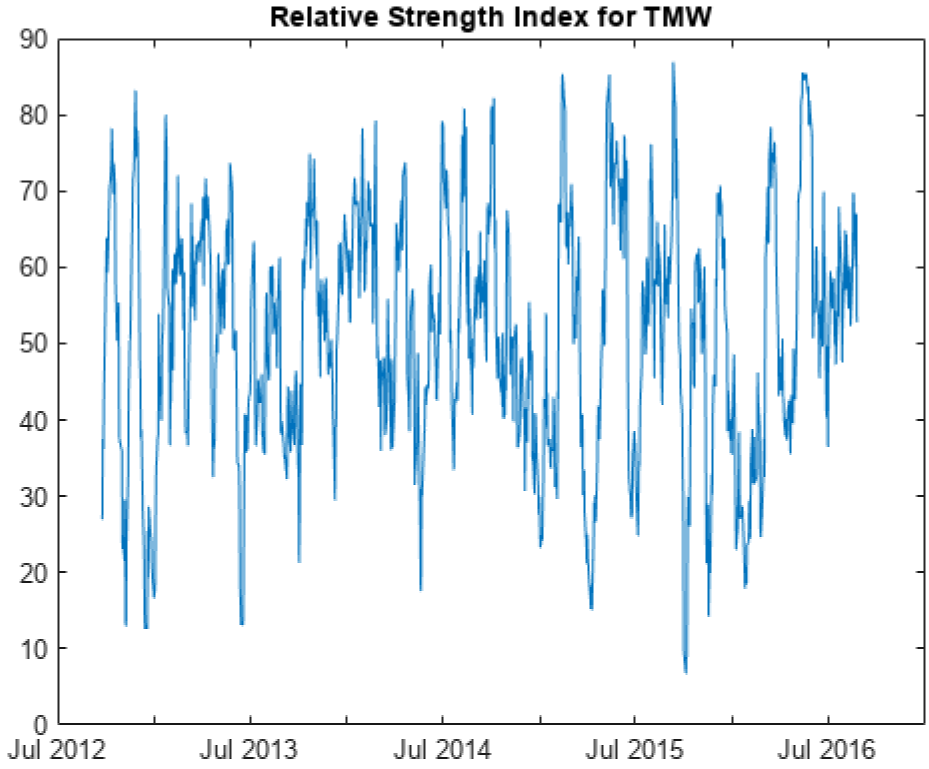
`index = rsindex( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Calculate the Relative Strength Index for a Data Series for a Stock

Load the file `SimulatedStock.mat`, which provides a timetable (`TMW`) for financial data for TMW stock.

```
load SimulatedStock.mat
index = rsindex(TMW);
plot(index.Time,index.RelativeStrengthIndex)
title('Relative Strength Index for TMW')
```



Input Arguments

`Data` — Data with closing prices

matrix | table | timetable

Data with closing prices, specified as a matrix, table, or timetable. For matrix input, `Data` is `M`-by-`1` with closing prices. Timetables and tables with `M` rows must contain a variable named `'Close'` (case insensitive).

Data Types: `double` | `table` | `timetable`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `index = rsindex(TMW,'WindowSize',10)`

`WindowSize` — Moving window size for relative strength index

`14` (default) | positive integer

Moving window size for relative strength index, specified as the comma-separated pair consisting of `'WindowSize'` and a scalar positive integer.

Data Types: `double`

Output Arguments

`index` — Relative strength index
matrix | table | timetable

Relative strength index, returned with the same number of rows (`M`) and the same type (matrix, table, or timetable) as the input `Data`.

More About

Relative Strength Index

Relative strength index is calculated by dividing the average of the gains by the average of the losses within a specified period.

```
RS = (average gains) / (average losses)
```

Version History

Introduced before R2006a

R2023a: `fints` support removed for `Data` input argument
Behavior changed in R2023a

`fints` object support for the `Data` input argument is removed.

R2022b: Support for negative price data
Behavior changed in R2022b

The `Data` input accepts negative prices.

References

[1] Murphy, John J. *Technical Analysis of the Futures Market.* New York Institute of Finance, 1986, pp. 295–302.

See Also

`timetable` | `table` | `negvolidx` | `posvolidx`

Topics
“Use Timetables in Finance” on page 11-7
“Convert Financial Time Series Objects (fints) to Timetables” on page 11-2

selectreturn

Portfolio configurations from 3-D efficient frontier

Syntax

```
PortConfigs = selectreturn(AllMean,AllCovariance,Target)
```

Description

`PortConfigs = selectreturn(AllMean,AllCovariance,Target)` returns the portfolio configurations (`PortConfigs`) for a target return given the average return and covariance for a rolling efficient frontier.

Input Arguments

`AllMean` — Expected asset returns used to generate each curve on surface

vector

Expected asset returns used to generate each curve on the surface, specified as a number of curves (`NCURVES`-by-`1` cell array), where each element is a `1`-by-`NASSETS` (number of assets) vector

Data Types: `double`

`AllCovariance` — Covariance matrix used to generate each curve on surface

cell array

Covariance matrix used to generate each curve on the surface, specified as an `NCURVES`-by-`1` cell array where each element is an `NASSETS`-by-`NASSETS` vector.

Data Types: `cell`

`Target` — Target return value for each curve in the frontier

numeric

Target return value for each curve in the frontier, specified as a numeric.

Data Types: `double`

Output Arguments

`PortConfigs` — Asset allocation weights needed to obtain target rate of return

matrix

Asset allocation weights needed to obtain the target rate of return, returned as an `NASSETS`-by-`NCURVES` matrix.

Version History

Introduced before R2006a

See Also

`targetreturn` | `frontier`

Topics

"Portfolio Construction Examples" on page 3-5
"Portfolio Optimization Functions" on page 3-3

setAssetList

Set up list of identifiers for assets

Syntax

```
obj = setAssetList(obj,AssetList)
obj = setAssetList(obj,'asset1','asset2',asset3',...)
```

Description

`obj = setAssetList(obj,AssetList)` sets up the list of identifiers for assets for `Portfolio`, `PortfolioCVaR`, or `PorfolioMAD` objects. For details on the respective workflows when using these different objects, see “Portfolio Object Workflow” on page 4-17, “PortfolioCVaR Object Workflow” on page 5-16, and “PortfolioMAD Object Workflow” on page 6-15.

`obj = setAssetList(obj,'asset1','asset2',asset3',...)` sets up a list of asset identifiers, specified as a comma-separated list of character vectors, a cell array of character vectors, or string array where each character vector or string is an asset identifier.

Note

- 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'`).

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'}
```

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`

Data Types: `object`

AssetList — List of assets

character vector | cell array of character vectors | string array

List of assets, specified using a character vector, cell array of character vectors, or string array where each character vector or string is an asset identifier.

Data Types: `char` | `cell` | `string`

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(AssetList);
```

- To clear an `AssetList`, call this function with `[]` or `{[]}`.

Version History

Introduced in R2011a

See Also

`estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierByRisk` | `estimateFrontierLimits`

Topics

"Common Operations on the Portfolio Object" on page 4-32
"Common Operations on the PortfolioCVaR Object" on page 5-29
"Common Operations on the PortfolioMAD Object" on page 6-28
"Portfolio Optimization Examples Using Financial Toolbox™" on page 4-152

setAssetMoments

Set moments (mean and covariance) of asset returns for Portfolio object

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. For details on the workflow, see “Portfolio Object Workflow” on page 4-17.

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 = 4×1

    0.0042
    0.0083
    0.0100
    0.0150


assetcovar = 4×4

    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
```

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

Data Types: object

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.
- If AssetCovar is not a symmetric positive semidefinite matrix, use nearcorr to create a positive semidefinite matrix for a correlation matrix.

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 `[]`.

Version History

Introduced in R2011a

See Also

`estimateAssetMoments` | `estimateFrontierByRisk` | `nearcorr` | `covarianceShrinkage` | `covarianceDenoising`

Topics

"Asset Returns and Moments of Asset Returns Using Portfolio Object" on page 4-41
"Portfolio Optimization Examples Using Financial Toolbox™" on page 4-152
"Portfolio Optimization Theory" on page 4-3

setBounds

Set up bounds for portfolio weights for portfolio

Syntax

```
obj = setBounds(obj,LowerBound)
obj = setBounds( ___ ,Name,Value)
```

```
obj = setBounds(obj,LowerBound,UpperBound)
obj = setBounds( ___ ,Name,Value)
```

Description

`obj = setBounds(obj,LowerBound)` sets up bounds for portfolio weights for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For details on the respective workflows when using these different objects, see “Portfolio Object Workflow” on page 4-17, “PortfolioCVaR Object Workflow” on page 5-16, and “PortfolioMAD Object Workflow” on page 6-15.

`obj = setBounds( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax, including `BoundType` as `'Simple'` or `'Conditional'`.

`obj = setBounds(obj,LowerBound,UpperBound)` sets up bounds for portfolio weights for portfolio objects with an additional option for `UpperBound`.

Given bound constraints `LowerBound` and `UpperBound` and `'Simple'` `BoundType`, every weight in a portfolio `Port` must satisfy the following:

```
LowerBound <= Port <= UpperBound
```

Given bound constraints `LowerBound` and `UpperBound`, and `'Conditional'` `BoundType`, every weight in a portfolio `Port` must satisfy the following:

```
Port = 0 or LowerBound <= Port <= UpperBound
```

`obj = setBounds( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax, including `BoundType` as `'Simple'` or `'Conditional'`.

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. Use `setBounds` to set the bound constraints for a balanced fund. Note that this sets default `'Simple'` `BoundType`, which enforces 0.5<=x1<=0.75, 0.25<=x2<=0.5.

```
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 to Define 'Conditional' BoundType Constraints for a Portfolio Object

Suppose you have the mean and covariance of the asset returns for a three asset portfolio:

```
AssetMean = [ 0.0101110; 0.0043532; 0.0137058 ];
AssetCovar = [ 0.00324625 0.00022983 0.00420395;
               0.00022983 0.00049937 0.00019247;
               0.00420395 0.00019247 0.00764097 ];
```

The following uses `setBounds` with `'Conditional'` `BoundType` (semicontinuous) constraints to set *xi* = `0` or `0.02 <= `*`xi`*` <= 0.5` for all *i* = 1,...`NumAssets`.

```
p = Portfolio('AssetMean', AssetMean, 'AssetCovar', AssetCovar);
p = setBounds(p, 0.02, 0.5,'BoundType', 'Conditional', 'NumAssets', 3)
```

```
p = 
  Portfolio with properties:

          BuyCost: []
         SellCost: []
     RiskFreeRate: []
        AssetMean: [3x1 double]
       AssetCovar: [3x3 double]
    TrackingError: []
     TrackingPort: []
         Turnover: []
      BuyTurnover: []
     SellTurnover: []
             Name: []
        NumAssets: 3
        AssetList: []
         InitPort: []
      AInequality: []
      bInequality: []
        AEquality: []
        bEquality: []
       LowerBound: [3x1 double]
       UpperBound: [3x1 double]
      LowerBudget: []
      UpperBudget: []
      GroupMatrix: []
       LowerGroup: []
       UpperGroup: []
           GroupA: []
           GroupB: []
       LowerRatio: []
       UpperRatio: []
     MinNumAssets: []
     MaxNumAssets: []
        BoundType: [3x1 categorical]
```

```
disp(p.LowerBound);
```

```
    0.0200
    0.0200
    0.0200
```

```
disp(p.UpperBound);
```

```
    0.5000
    0.5000
    0.5000
```

```
disp(p.BoundType);
```

```
     conditional
     conditional
     conditional
```

Set Bound Constraints to Define Mixed BoundTypes for a Portfolio Object

Suppose you have the mean and covariance of the asset returns for a three asset portfolio:

```
AssetMean = [ 0.0101110; 0.0043532; 0.0137058 ];
AssetCovar = [ 0.00324625 0.00022983 0.00420395;
               0.00022983 0.00049937 0.00019247;
               0.00420395 0.00019247 0.00764097 ];
```

The following uses `setBounds` with both `'Simple'` and `'Conditional'` `BoundType` constraints for all *i* = 1,...`NumAssets`.

```
p = Portfolio('AssetMean', AssetMean, 'AssetCovar', AssetCovar);
p = setBounds(p, 0.1, 0.5, 'BoundType',["simple"; "conditional"; "conditional"])
```

```
p = 
  Portfolio with properties:

          BuyCost: []
         SellCost: []
     RiskFreeRate: []
        AssetMean: [3x1 double]
       AssetCovar: [3x3 double]
    TrackingError: []
     TrackingPort: []
```

```
          Turnover: []
       BuyTurnover: []
      SellTurnover: []
              Name: []
         NumAssets: 3
         AssetList: []
          InitPort: []
       AInequality: []
       bInequality: []
         AEquality: []
         bEquality: []
        LowerBound: [3x1 double]
        UpperBound: [3x1 double]
       LowerBudget: []
       UpperBudget: []
       GroupMatrix: []
        LowerGroup: []
        UpperGroup: []
            GroupA: []
            GroupB: []
        LowerRatio: []
        UpperRatio: []
      MinNumAssets: []
      MaxNumAssets: []
         BoundType: [3x1 categorical]
```

```
disp(p.LowerBound);

    0.1000
    0.1000
    0.1000

disp(p.UpperBound);

    0.5000
    0.5000
    0.5000

disp(p.BoundType);

     simple
     conditional
     conditional
```

You can use supply lower and upper bounds as vectors, which defines different values for each asset. The following have -0.8 <= $x1$ <= 0.2; $x2$ = 0 or 0.1 <= $x2$ <= 0.5; $x3$ = 0 or 0.1 <= $x3$ <=0.5. Note that as `'Simple'` `BoundType`, the assets can be held as short or long positions. However, when using `'Conditional'` `BoundType`, the assets can only be long positions.

```
p = setBounds(p, [-0.8, 0.1, 0.1], [-0.2,0.5,0.5], 'BoundType',["simple"; "conditional"; "condit
disp(p.LowerBound);

   -0.8000
    0.1000
    0.1000

disp(p.UpperBound);
```

```
   -0.2000
    0.5000
    0.5000
```

```
disp(p.BoundType);
```

```
     simple
     conditional
     conditional
```

Set MinNumAssets Constraint for a Portfolio Object

Set the minimum cardinality constraint for a three-asset portfolio for which you have the mean and covariance values of the asset returns.

```
AssetMean = [ 0.0101110; 0.0043532; 0.0137058 ];
AssetCovar = [ 0.00324625 0.00022983 0.00420395;
               0.00022983 0.00049937 0.00019247;
               0.00420395 0.00019247 0.00764097 ];

p = Portfolio('AssetMean', AssetMean, 'AssetCovar', AssetCovar);
p = setDefaultConstraints(p);
```

When working with a `Portfolio` object, the `setMinMaxNumAssets` function enables you to set up limits on the number of assets invested. These limits are also known as cardinality constraints. When managing a portfolio, it is common that you want to invest in at least a certain number of assets. In addition, you should also clearly define the weight requirement for each invested asset. You can do this using `setBounds` with a `'Conditional'` `BoundType`. If you do not specify a `'Conditional'` `BoundType`, the optimizer cannot understand which assets are invested assets and cannot formulate the `MinNumAssets` constraint.

The following example specifies that at least two assets should be invested and the investments should be greater than 16%.

```
p = setMinMaxNumAssets(p, 2, []);
p = setBounds(p, 0.16, 'BoundType', 'conditional');
```

Use `estimateFrontierByReturn` to estimate optimal portfolios with targeted portfolio returns.

```
pwgt = estimateFrontierByReturn(p,[ 0.008, 0.01 ])
```

```
pwgt = 3×2

    0.2861    0.3967
    0.5001    0.2437
    0.2138    0.3595
```

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 to Define 'Conditional' BoundType 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 with semicontinuous constraints, use `setBounds` with `'Conditional'` `BoundType` constraints to set *xi* = `0.25` or `0.5` <= *`xi`* <= `0.5 or 0.75` for all *i* = 1,...`NumAssets`.

```
lb = [ 0.5; 0.25 ];
ub = [ 0.75; 0.5 ];

p = PortfolioCVaR;
p = setBounds(p,lb,ub,'BoundType',["conditional"; "conditional"]);
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
```

Set Bound Constraints to Define 'Conditional' BoundType 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 with semicontinuous constraints, use `setBounds` with `'Conditional'` `BoundType` constraints to set *xi* = `0.25` or `0.5 <= xi <= 0.5 or 0.75` for all *i* = 1,...`NumAssets`.

```
lb = [ 0.5; 0.25 ];
ub = [ 0.75; 0.5 ];

p = PortfolioMAD;
p = setBounds(p,lb,ub,'BoundType',["conditional"; "conditional"]);
disp(p.NumAssets);

     2

disp(p.LowerBound);

    0.5000
    0.2500

disp(p.UpperBound);

    0.7500
    0.5000
```

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`

Data Types: `object`

`LowerBound` — Lower-bound weight for each asset
scalar | vector

Lower-bound weight for each asset, specified as a scalar or 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`, `PortfolioCVaR`, or `PortfolioMAD` object are cleared and set to `[]`.
- If `LowerBound` or `UpperBound` are specified as scalars and `NumAssets` exists or can be computed, 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.
- If `'Conditional'``BoundType` is specified, the `LowerBound` cannot be a negative value.

Data Types: `double`

`UpperBound` — Upper-bound weight for each asset
scalar | vector

(Optional) Upper-bound weight for each asset, specified as a scalar or 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`, `PortfolioCVaR`, or `PortfolioMAD` object are cleared and set to `[]`.
- If `LowerBound` or `UpperBound` are specified as scalars and `NumAssets` exists or can be computed, 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.
- If `'Conditional'``BoundType` is specified, the `UpperBound` cannot be a negative value.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `obj = setBounds(p,0.02,'BoundType','Conditional');`

BoundType — Type of bounds for each asset weight
'Simple' (default) | character vector with value 'Simple' or 'Conditional' | string with value "Simple" or "Conditional" | cell array of character vectors with values 'Simple' or 'Conditional' | string array with values "Simple" or "Conditional"

Type of bounds for each asset weight, specified as the comma-separated pair consisting of 'BoundType' and a scalar character vector or string with a value of 'Simple' or 'Conditional' or a cell array of character vectors with values of 'Simple' or 'Conditional'.

- 'Simple' is LowerBound <= AssetWeight <= UpperBound.
- 'Conditional' is LowerBound <= AssetWeight <= UpperBound or AssetWeight = 0.

Warning If you specify the Bound range to be inclusive of zero (using either a ‘Simple’ or ‘Conditional’ BoundType), when you use setMinMaxNumAssets to specify the MinNumAssets constraint, and then use one of the estimate functions, it is ambiguous for the optimizer to define the minimum requirement for an allocated asset. In this case, the optimizer considers that an asset with zero weight is a valid allocated asset and the optimization proceeds, but with the warning that the allocation has less than the MinNumAssets required. For more information, see “Troubleshooting for Setting 'Conditional' BoundType, MinNumAssets, and MaxNumAssets Constraints” on page 4-139.

Data Types: char | cell | string

NumAssets — Number of assets in portfolio
1 (default) | scalar numeric

Number of assets in portfolio, specified as the comma-separated pair consisting of 'NumAssets' and a scalar numeric value.

Note NumAssets cannot be used to change the dimension of a Portfolio, PortfolioCVaR, or PortfolioMAD object.

- If either LowerBound or UpperBound are input as empties with [], the corresponding attributes in the Portfolio, PortfolioCVaR, or PortfolioMAD 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

Updated portfolio object, returned as a Portfolio, PortfolioCVaR, or PortfolioMAD object.

Tips

You can also use dot notation to set up the bounds for portfolio weights.

```
 obj = obj.setBounds(LowerBound, UpperBound, Name,Value);
```

If any of `LowerBound`, `UpperBound`, or `BoundType`are input as empties with `[ ]`, the corresponding attributes in the portfolio object are cleared and set to `[ ]`. If `BoundType` is cleared as `[ ]`, the bound type defaults to `'Simple'`.

```
 p = setBounds(p, LowerBound, [ ], 'BoundType',[ ]);
```

To reset a portfolio object to be a continuous problem, run the following:

```
p = setMinMaxNumAssets(p, [],[]);
p = setBounds(p, p.LowerBound, p.UpperBound, 'BoundType', 'Simple');
```

Version History

Introduced in R2011a

See Also

`getBounds` | `setMinMaxNumAssets` | `setSolverMINLP` | `estimateAssetMoments` | `estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierByRisk` | `estimateFrontierLimits` | `estimateMaxSharpeRatio` | `estimatePortSharpeRatio` | `estimatePortMoments` | `estimatePortReturn` | `estimatePortRisk`

Topics

“Working with 'Simple' Bound Constraints Using Portfolio Object” on page 4-61
“Working with 'Simple' Bound Constraints Using PortfolioCVaR Object” on page 5-54
“Working with 'Simple' Bound Constraints Using PortfolioMAD Object” on page 6-52
“Working with 'Conditional' BoundType, MinNumAssets, and MaxNumAssets Constraints Using Portfolio Objects” on page 4-78
“Working with 'Conditional' BoundType, MinNumAssets, and MaxNumAssets Constraints Using PortfolioCVaR Objects” on page 5-69
“Working with 'Conditional' BoundType, MinNumAssets, and MaxNumAssets Constraints Using PortfolioMAD Objects” on page 6-67
“Troubleshooting for Setting 'Conditional' BoundType, MinNumAssets, and MaxNumAssets Constraints” on page 4-139
“Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
“Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-183
“Portfolio Set for Optimization Using Portfolio Objects” on page 4-8
“Portfolio Set for Optimization Using PortfolioCVaR Object” on page 5-8
“Portfolio Set for Optimization Using PortfolioMAD Object” on page 6-7

setBudget

Set up budget constraints for portfolio

Syntax

```
obj = setBudget(obj,LowerBudget)
obj = setBudget(obj,LowerBudget,UpperBudget)
```

Description

`obj = setBudget(obj,LowerBudget)` sets up budget constraints for `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` objects. For details on the respective workflows when using these different objects, see “Portfolio Object Workflow” on page 4-17, “PortfolioCVaR Object Workflow” on page 5-16, and “PortfolioMAD Object Workflow” on page 6-15.

`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
```

Control the Allocation of a Risk-Free Asset Using setBudget

Define mean and covariance of risk asset returns.

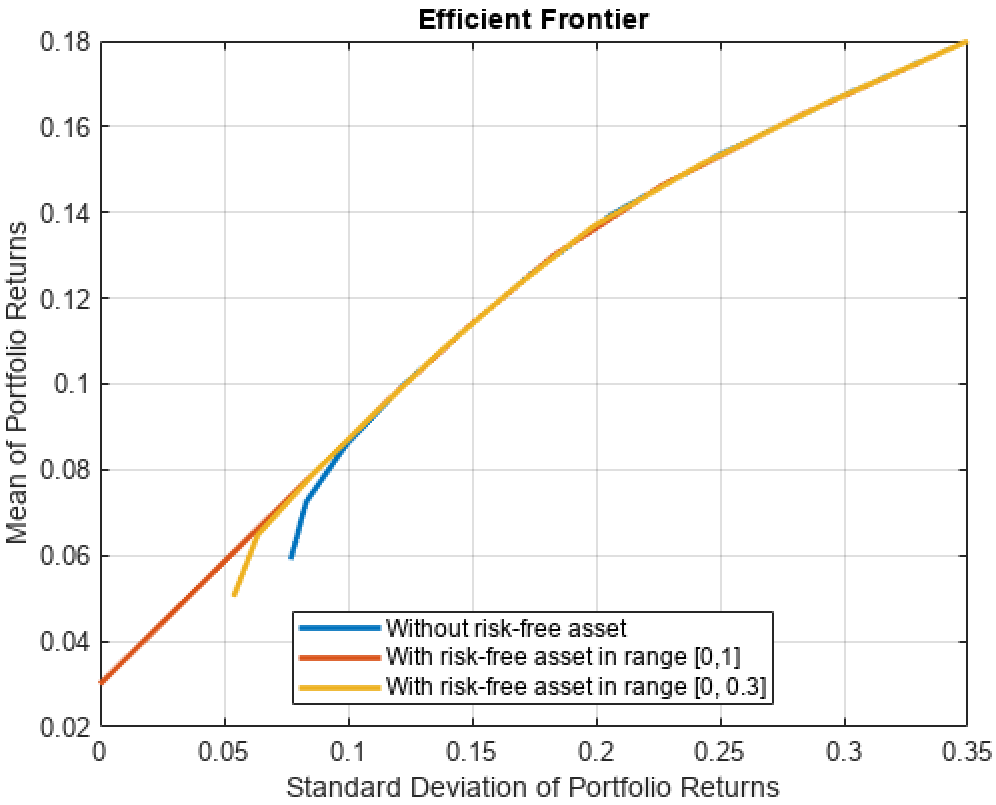
```
m = [ 0.05; 0.1; 0.12; 0.18; ];
C = [ 0.0064 0.00408 0.00192 0,;
    0.00408 0.0289 0.0204 0.0119,;
    0.00192 0.0204 0.0576 0.0336,;
   0 0.0119 0.0336 0.1225];
```

Create a `Portfolio` object defining the risk-free rate.

```
p = Portfolio('RiskFreeRate',0.03, 'assetmean', m, 'assetcovar', C, ...
'lowerbudget', 1, 'upperbudget', 1, 'lowerbound', 0);
```

Create multiple `Portfolio` objects with different budgets on risky assets. By defining the risky assets, you can control how much is invested in a risk-free asset.

```
p = setBudget(p, 1, 1);    % allow 0% risk-free asset allocation, meaning fully invested in risky
p1 = setBudget(p, 0, 1);   % allow 0 to 100% risk-free asset allocation
p2 = setBudget(p, 0.7, 1);  % allow 0 to 30% risk-free asset allocation

plotFrontier(p); hold on;
plotFrontier(p1);hold on;
plotFrontier(p2);
legend('Without risk-free asset', 'With risk-free asset in range [0,1]', 'With risk-free asset in
```



`setBudget` defines the bound for total weights for the allocated risky assets, and the remaining is automatically the bound for a risk-free asset. Use `setBudget` to control the level of allowed allocation to a risk-free asset. For additional information on using `setBudget` with a risk-free asset, see "Leverage in Portfolio Optimization with a Risk-Free Asset" on page 4-210.

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`

Data Types: `object`

`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);
```

Version History

Introduced in R2011a

See Also

`getBudget`

Topics

“Working with Budget Constraints Using Portfolio Object” on page 4-64
“Working with Budget Constraints Using PortfolioCVaR Object” on page 5-57
“Working with Budget Constraints Using PortfolioMAD Object” on page 6-55
“Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
“Leverage in Portfolio Optimization with a Risk-Free Asset” on page 4-210
“Portfolio Set for Optimization Using Portfolio Objects” on page 4-8
“Portfolio Set for Optimization Using PortfolioCVaR Object” on page 5-8
“Portfolio Set for Optimization Using PortfolioMAD Object” on page 6-7

setCosts

Set up proportional transaction costs for portfolio

Syntax

```
obj = setCosts(obj,BuyCost)
```

```
obj = setCosts(obj,BuyCost,SellCost,InitPort,NumAssets)
```

Description

`obj = setCosts(obj,BuyCost)` sets up proportional transaction costs for `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` objects. For details on the respective workflows when using these different objects, see "Portfolio Object Workflow" on page 4-17, "PortfolioCVaR Object Workflow" on page 5-16, and "PortfolioMAD Object Workflow" on page 6-15.

`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

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
```

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`

Data Types: `object`

`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.

Version History

Introduced in R2011a

See Also

getCosts | setInitPort

Topics

“Working with Transaction Costs” on page 4-53
“Working with Transaction Costs” on page 5-46
“Working with Transaction Costs” on page 6-44
“Portfolio Analysis with Turnover Constraints” on page 4-204
“Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
“Portfolio Set for Optimization Using Portfolio Objects” on page 4-8
“Portfolio Set for Optimization Using PortfolioCVaR Object” on page 5-8
“Portfolio Set for Optimization Using PortfolioMAD Object” on page 6-7

setDefaultConstraints

Set up portfolio constraints with nonnegative weights that sum to 1

Syntax

```
obj = setDefaultConstraints(obj)
obj = setDefaultConstraints(obj,NumAssets)
```

Description

`obj = setDefaultConstraints(obj)` sets up portfolio constraints with nonnegative weights that sum to 1 for `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` objects. For details on the respective workflows when using these different objects, see “Portfolio Object Workflow” on page 4-17, “PortfolioCVaR Object Workflow” on page 5-16, and “PortfolioMAD Object Workflow” on page 6-15.

`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: []
     MinNumAssets: []
     MaxNumAssets: []
        BoundType: [20x1 categorical]
```

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: []
        MinNumAssets: []
        MaxNumAssets: []
           BoundType: [20x1 categorical]
```

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: []
    MinNumAssets: []
    MaxNumAssets: []
       BoundType: [20x1 categorical]
```

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`

Data Types: `object`

`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 `[]`.

Version History

Introduced in R2011a

See Also

`setBounds` | `getBounds` | `setBudget`

Topics

“Setting Default Constraints for Portfolio Weights Using Portfolio Object” on page 4-57
“Setting Default Constraints for Portfolio Weights Using PortfolioCVaR Object” on page 5-50
“Setting Default Constraints for Portfolio Weights Using PortfolioMAD Object” on page 6-48
“Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
“Portfolio Set for Optimization Using Portfolio Objects” on page 4-8
“Portfolio Set for Optimization Using PortfolioCVaR Object” on page 5-8
“Portfolio Set for Optimization Using PortfolioMAD Object” on page 6-7

setEquality

Set up linear equality constraints for portfolio weights

Syntax

```
obj= setEquality(obj,AEquality,bEquality)
```

Description

`obj= setEquality(obj,AEquality,bEquality)` sets up linear equality constraints for portfolio weights for `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` objects. For details on the respective workflows when using these different objects, see “Portfolio Object Workflow” on page 4-17, “PortfolioCVaR Object Workflow” on page 5-16, and “PortfolioMAD Object Workflow” on page 6-15.

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 = 1×5

     1     1     1     0     0


bEquality = 0.5000
```

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`

Data Types: `object`

`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.

Version History

Introduced in R2011a

See Also

addEquality | getEquality

Topics

“Working with Linear Equality Constraints Using Portfolio Object” on page 4-72
“Working with Linear Equality Constraints Using PortfolioCVaR Object” on page 5-65
“Working with Linear Equality Constraints Using PortfolioMAD Object” on page 6-63
“Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
“Portfolio Set for Optimization Using Portfolio Objects” on page 4-8
“Portfolio Set for Optimization Using PortfolioCVaR Object” on page 5-8

"Portfolio Set for Optimization Using PortfolioMAD Object" on page 6-7

setGroupRatio

Set up group ratio constraints for portfolio weights

Syntax

```
obj = setGroupRatio(obj,GroupA,GroupB,LowerRatio)
obj = setGroupRatio(___,UpperRatio)
```

Description

`obj = setGroupRatio(obj,GroupA,GroupB,LowerRatio)` sets up group ratio constraints for portfolio weights for `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` objects. For details on the respective workflows when using these different objects, see “Portfolio Object Workflow” on page 4-17, “PortfolioCVaR Object Workflow” on page 5-16, and “PortfolioMAD Object Workflow” on page 6-15.

`obj = setGroupRatio(___,UpperRatio)` sets up group ratio constraints for portfolio weights for portfolio objects with an additional optional argument for `UpperRatio`.

Given base and comparison group matrices `GroupA` and `GroupB` and `LowerRatio` or `UpperRatio` bounds, group ratio constraints require any portfolio in `Port` to satisfy the following:

```
(GroupB * Port) .* LowerRatio <= GroupA * Port <= (GroupB * Port) .* UpperRatio
```

Caution This collection of constraints usually requires that portfolio weights be nonnegative and that the products `GroupA * Port` and `GroupB * Port` are always nonnegative. Although negative portfolio weights and non-Boolean group ratio matrices are supported, use with caution.

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 nonfinancial companies (assets 4-6). Group ratio constraints can be set with:

```
GA = [ true true true false false false ];    % financial companies
GB = [ false false false true true true ];    % nonfinancial companies
p = Portfolio;
p = setGroupRatio(p, GA, GB, [], 0.5);

disp(p.NumAssets);

     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 nonfinancial companies (assets 4-6). Group ratio constraints can be set with:

```
GA = [ true true true false false false ];    % financial companies
GB = [ false false false true true true ];    % nonfinancial companies
p = PortfolioCVaR;
p = setGroupRatio(p, GA, GB, [], 0.5);

disp(p.NumAssets);
```

```
     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 nonfinancial companies (assets 4-6). Group ratio constraints can be set with:

```
GA = [ true true true false false false ];    % financial companies
GB = [ false false false true true true ];    % nonfinancial companies
p = PortfolioMAD;
p = setGroupRatio(p, GA, GB, [], 0.5);

disp(p.NumAssets);
```

```
     6
```

```
disp(p.GroupA);
```

```
     1     1     1     0     0     0
```

```
disp(p.GroupB);
```

```
     0     0     0     1     1     1
```

```
disp(p.UpperRatio);

    0.5000
```

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

Data Types: object

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 | logical

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 | logical

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.

Version History

Introduced in R2011a

See Also

addGroupRatio | getGroupRatio

Topics

“Working with Group Ratio Constraints Using Portfolio Object” on page 4-69
“Working with Group Constraints Using PortfolioCVaR Object” on page 5-59
“Working with Group Constraints Using PortfolioMAD Object” on page 6-57
“Portfolio Optimization Examples Using Financial Toolbox™ ” on page 4-152
“Portfolio Set for Optimization Using Portfolio Objects” on page 4-8

"Portfolio Set for Optimization Using PortfolioCVaR Object" on page 5-8
"Portfolio Set for Optimization Using PortfolioMAD Object" on page 6-7

setGroups

Set up group constraints for portfolio weights

Syntax

```
obj = setGroups(obj,GroupMatrix,LowerGroup)
obj = setGroups(obj,GroupMatrix,LowerGroup,UpperGroup)
```

Description

`obj = setGroups(obj,GroupMatrix,LowerGroup)` sets up group constraints for portfolio weights for `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` objects. For details on the respective workflows when using these different objects, see "Portfolio Object Workflow" on page 4-17, "PortfolioCVaR Object Workflow" on page 5-16, and "PortfolioMAD Object Workflow" on page 6-15.

`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
```

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

Data Types: object

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`.

Version History

Introduced in R2011a

See Also

`getGroups` | `addGroups`

Topics

"Working with Group Constraints Using Portfolio Object" on page 4-66
"Working with Group Constraints Using PortfolioCVaR Object" on page 5-59
"Working with Group Constraints Using PortfolioMAD Object" on page 6-57
"Constraint Specification Using a Portfolio Object" on page 3-19
"Portfolio Optimization Examples Using Financial Toolbox™" on page 4-152
"Portfolio Set for Optimization Using Portfolio Objects" on page 4-8
"Portfolio Set for Optimization Using PortfolioCVaR Object" on page 5-8
"Portfolio Set for Optimization Using PortfolioMAD Object" on page 6-7

setInequality

Set up linear inequality constraints for portfolio weights

Syntax

```
obj = setInequality(obj,AInequality,bInequality)
```

Description

`obj = setInequality(obj,AInequality,bInequality)` sets up linear inequality constraints for portfolio weights for `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` objects. For details on the respective workflows when using these different objects, see “Portfolio Object Workflow” on page 4-17, “PortfolioCVaR Object Workflow” on page 5-16, and “PortfolioMAD Object Workflow” on page 6-15.

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
```

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

Data Types: object

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.

Version History

Introduced in R2011a

See Also

getInequality | addInequality

Topics

“Working with Linear Inequality Constraints Using Portfolio Object” on page 4-75
“Working with Linear Inequality Constraints Using PortfolioCVaR Object” on page 5-67
“Working with Linear Inequality Constraints Using PortfolioMAD Object” on page 6-65
“Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
“Portfolio Set for Optimization Using Portfolio Objects” on page 4-8
“Portfolio Set for Optimization Using PortfolioCVaR Object” on page 5-8
“Portfolio Set for Optimization Using PortfolioMAD Object” on page 6-7

setInitPort

Set up initial or current portfolio

Syntax

```
obj = setInitPort(obj,InitPort)
obj = setInitPort(obj,InitPort,NumAssets)
```

Description

`obj = setInitPort(obj,InitPort)` sets up initial or current portfolio for `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` objects. For details on the respective workflows when using these different objects, see “Portfolio Object Workflow” on page 4-17, “PortfolioCVaR Object Workflow” on page 5-16, and “PortfolioMAD Object Workflow” on page 6-15.

`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
```

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`

Data Types: `object`

`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 function with an empty argument `[]` for `InitPort`.

Version History

Introduced in R2011a

See Also

`setTurnover` | `setCosts`

Topics

“Common Operations on the Portfolio Object” on page 4-32
“Common Operations on the PortfolioCVaR Object” on page 5-29
“Common Operations on the PortfolioMAD Object” on page 6-28
“Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
“Portfolio Set for Optimization Using Portfolio Objects” on page 4-8
“Portfolio Set for Optimization Using PortfolioCVaR Object” on page 5-8
“Portfolio Set for Optimization Using PortfolioMAD Object” on page 6-7

setMinMaxNumAssets

Set cardinality constraints on the number of assets invested in a portfolio

Syntax

```
obj = setMinMaxNumAssets(obj,MinNumAssets,MaxNumAssets)
```

Description

`obj = setMinMaxNumAssets(obj,MinNumAssets,MaxNumAssets)` sets cardinality constraints for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object.

`MinNumAssets` and `MaxNumAssets` are the minimum and maximum number of assets invested in the portfolio, respectively. The total number of allocated assets satisfying the Bound constraints is between [`MinNumAssets`, `MaxNumAssets`]. For details on the respective workflows when using these different objects, see “Portfolio Object Workflow” on page 4-17, “PortfolioCVaR Object Workflow” on page 5-16, and “PortfolioMAD Object Workflow” on page 6-15.

Examples

Set `MaxNumAssets` Constraint for a `Portfolio` Object

Set the maximum cardinality constraint for a three-asset portfolio for which you have the mean and covariance values of the asset returns.

```
AssetMean = [ 0.0101110; 0.0043532; 0.0137058 ];
AssetCovar = [ 0.00324625 0.00022983 0.00420395;
               0.00022983 0.00049937 0.00019247;
               0.00420395 0.00019247 0.00764097 ];

p = Portfolio('AssetMean', AssetMean, 'AssetCovar', AssetCovar);
p = setDefaultConstraints(p);
```

When working with a `Portfolio` object, the `setMinMaxNumAssets` function enables you to set up the limits on the number of assets invested. Use `setMinMaxNumAssets` to limit the total number of allocated assets to no more than two.

```
 p = setMinMaxNumAssets(p, [], 2);
```

Use `estimateFrontierByReturn` to estimate optimal portfolios with targeted portfolio returns.

```
pwgt = estimateFrontierByReturn(p,[ 0.008, 0.01 ])
```

```
pwgt = 3×2

         0         0
    0.6101    0.3962
    0.3899    0.6038
```

Set MinNumAssets Constraint for a Portfolio Object

Set the minimum cardinality constraint for a three-asset portfolio for which you have the mean and covariance values of the asset returns.

```
AssetMean = [ 0.0101110; 0.0043532; 0.0137058 ];
AssetCovar = [ 0.00324625 0.00022983 0.00420395;
               0.00022983 0.00049937 0.00019247;
               0.00420395 0.00019247 0.00764097 ];

p = Portfolio('AssetMean', AssetMean, 'AssetCovar', AssetCovar);
p = setDefaultConstraints(p);
```

When working with a Portfolio object, the setMinMaxNumAssets function enables you to set up limits on the number of assets invested. These limits are also known as cardinality constraints. When managing a portfolio, it is common that you want to invest in at least a certain number of assets. In addition, you should also clearly define the weight requirement for each invested asset. You can do this using setBounds with a 'Conditional' BoundType. If you do not specify a 'Conditional' BoundType, the optimizer cannot understand which assets are invested assets and cannot formulate the MinNumAssets constraint.

The following example specifies that at least two assets should be invested and the investments should be greater than 16%.

```
p = setMinMaxNumAssets(p, 2, []);
p = setBounds(p, 0.16, 'BoundType', 'conditional');
```

Use estimateFrontierByReturn to estimate optimal portfolios with targeted portfolio returns.

```
pwgt = estimateFrontierByReturn(p,[ 0.008, 0.01 ])
```

```
pwgt = 3×2

    0.2861    0.3967
    0.5001    0.2437
    0.2138    0.3595
```

Set 'Conditional' BoundType, MinNumAssets, and MaxNumAssets Constraints for a Portfolio Object

Set the minimum and maximum cardinality constraints and a 'Conditional' BoundType for a three-asset portfolio for which you have the mean and covariance values of the asset returns.

```
AssetMean = [ 0.0101110; 0.0043532; 0.0137058 ];
AssetCovar = [ 0.00324625 0.00022983 0.00420395;
               0.00022983 0.00049937 0.00019247;
               0.00420395 0.00019247 0.00764097 ];

p = Portfolio('AssetMean', AssetMean, 'AssetCovar', AssetCovar);
p = setDefaultConstraints(p);
```

When working with a Portfolio object, the setMinMaxNumAssets function enables you to set up the limits on the number of assets invested. The following example specifies that exactly two assets should be invested using `setMinMaxNumAssets` and the investment should be equally allocated among the two assets using `setBounds`.

```
p = setMinMaxNumAssets(p, 2, 2);
p = setBounds(p, 0.5, 0.5, 'BoundType', 'conditional');
```

Use `estimateFrontierByReturn` to estimate optimal portfolios with targeted portfolio returns.

```
pwgt = estimateFrontierByReturn(p,[ 0.008, 0.01 ])
```

```
pwgt = 3×2

         0    0.5000
    0.5000         0
    0.5000    0.5000
```

Set 'Conditional' BoundType, MinNumAssets, and MaxNumAssets Constraints for a PortfolioCVaR Object

Suppose you have a universe of 12 stocks where you want to find the optimal portfolios with targeted returns and you want to set semicontinuous and cardinality constraints for the portfolio.

```
load CAPMuniverse
p = PortfolioCVaR('AssetList',Assets(1:12));
p = simulateNormalScenariosByData(p, Data(:,1:12), 20000 ,'missingdata',true);
p = setProbabilityLevel(p, 0.80);
```

When working with a `PortfolioCVaR` object, the `setMinMaxNumAssets` function enables you to set up the limits on the number of assets invested. The following example specifies that a minimum of five assets and a maximum of 10 assets should be invested using `setMinMaxNumAssets` and the investments should be greater than 4% and less than 45% using `setBounds`.

```
p = setMinMaxNumAssets(p, 5, 10);
p = setBounds(p, 0.04, 0.45, 'BoundType', 'conditional');
```

Use `estimateFrontierByReturn` to estimate optimal portfolios with targeted portfolio returns.

```
pwgt = estimateFrontierByReturn(p,[ 0.00026, 0.00038 ])
```

```
pwgt = 12×2

    0.0400    0.0400
    0.0000         0
         0         0
         0         0
         0         0
    0.0496    0.0786
         0    0.0400
    0.0400    0.0400
         0         0
    0.0400    0.0400
      ⋮
```

Set 'Conditional' BoundType, MinNumAssets, and MaxNumAssets Constraints for a PortfolioMAD Object

Suppose you have a universe of 12 stocks where you want to find the optimal portfolios with targeted returns and you want to set semicontinuous and cardinality constraints for the portfolio.

```
load CAPMuniverse
p = PortfolioMAD('AssetList',Assets(1:12));
p = simulateNormalScenariosByData(p, Data(:,1:12), 20000 ,'missingdata',true);
```

When working with a `PortfolioMAD` object, the `setMinMaxNumAssets` function enables you to set up the limits on the number of assets invested. The following example specifies that a minimum of five assets and a maximum of 10 assets should be invested using `setMinMaxNumAssets` and the investments should be greater than 4% and less than 45% using `setBounds`.

```
p = setMinMaxNumAssets(p, 5, 10);
p = setBounds(p, 0.04, 0.45, 'BoundType', 'conditional');
```

Use `estimateFrontierByReturn` to estimate optimal portfolios with targeted portfolio returns.

```
pwgt = estimateFrontierByReturn(p,[ 0.00026, 0.00038 ])
```

```
pwgt = 12×2

    0.0400    0.0400
         0         0
         0         0
         0         0
         0         0
    0.0507    0.0786
    0.0400    0.0400
    0.0400    0.0400
         0         0
    0.0400    0.0400
      ⋮
```

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`

Data Types: `object`

MinNumAssets — Minimum number of assets allocated in portfolio

scalar numeric

Minimum number of assets allocated in the portfolio, specified using a scalar numeric.

Data Types: `double`

`MaxNumAssets` — Maximum number of assets allocated in portfolio
scalar numeric

Maximum number of assets allocated in the portfolio, specified using a scalar numeric.

Data Types: `double`

Output Arguments

`obj` — Updated portfolio object
object for portfolio

Updated portfolio object, returned as a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object.

Tips

- You can also use dot notation to set up a list of identifiers for assets.

  ```
  obj = obj.setMinMaxNumAssets(MinNumAssets,MaxNumAssets);
  ```
- Specifying empty values (`[[]`) for `MinNumAssets` and `MaxNumAsssets` removes limit constraints from the `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object.

Version History

Introduced in R2018b

See Also

`setBounds` | `setSolverMINLP` | `estimateAssetMoments` | `estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierByRisk` | `estimateFrontierLimits` | `estimateMaxSharpeRatio` | `estimatePortSharpeRatio` | `estimatePortMoments` | `estimatePortReturn` | `estimatePortRisk`

Topics

"Working with 'Conditional' BoundType, MinNumAssets, and MaxNumAssets Constraints Using Portfolio Objects" on page 4-78
"Working with 'Conditional' BoundType, MinNumAssets, and MaxNumAssets Constraints Using PortfolioCVaR Objects" on page 5-69
"Working with 'Conditional' BoundType, MinNumAssets, and MaxNumAssets Constraints Using PortfolioMAD Objects" on page 6-67
"Troubleshooting for Setting 'Conditional' BoundType, MinNumAssets, and MaxNumAssets Constraints" on page 4-139
"Common Operations on the Portfolio Object" on page 4-32
"Portfolio Optimization Examples Using Financial Toolbox™" on page 4-152
"Portfolio Optimization with Semicontinuous and Cardinality Constraints" on page 4-183
"Mixed-Integer Quadratic Programming Portfolio Optimization: Problem-Based"
"Role of Convexity in Portfolio Problems" on page 4-148
"Portfolio Set for Optimization Using Portfolio Objects" on page 4-8
"Portfolio Set for Optimization Using PortfolioCVaR Object" on page 5-8

"Portfolio Set for Optimization Using PortfolioMAD Object" on page 6-7

setOneWayTurnover

Set up one-way portfolio turnover constraints

Syntax

```
obj = setOneWayTurnover(obj,BuyTurnover)
obj = setOneWayTurnover(obj,BuyTurnover,SellTurnover,InitPort,NumAssets)
```

Description

`obj = setOneWayTurnover(obj,BuyTurnover)` sets up one-way portfolio turnover constraints for `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` objects. For details on the respective workflows when using these different objects, see “Portfolio Object Workflow” on page 4-17, “PortfolioCVaR Object Workflow” on page 5-16, and “PortfolioMAD Object Workflow” on page 6-15.

`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
```

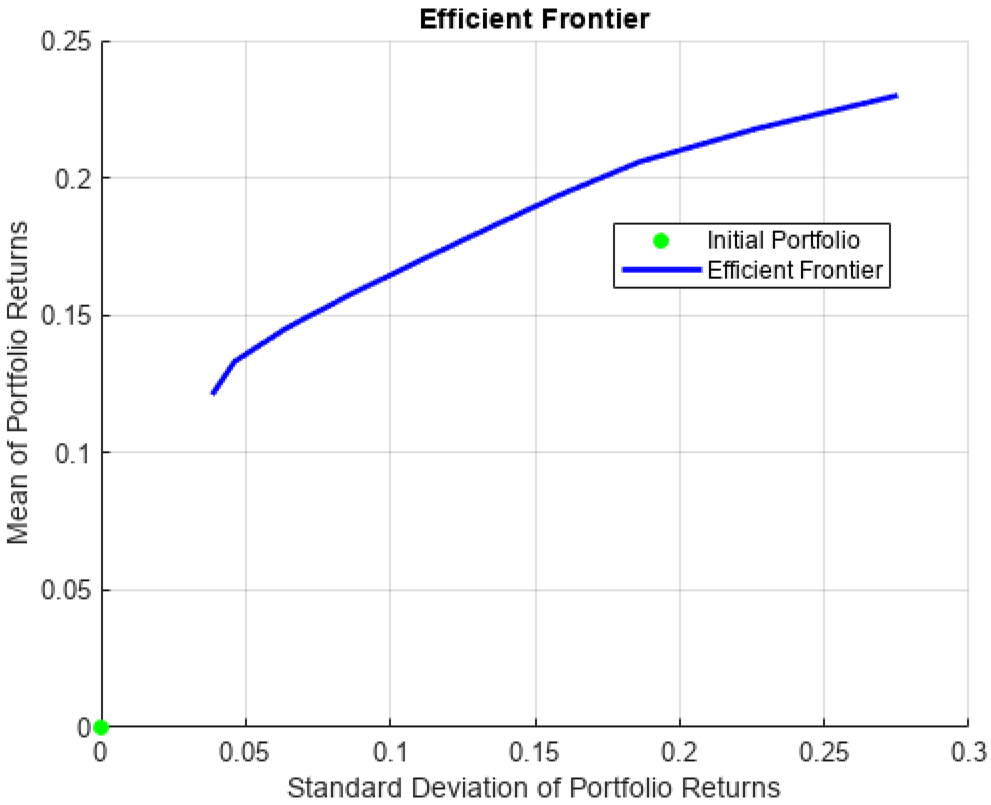
To set this constraint, use `setTurnover`.

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
```

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`

Data Types: `object`

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`

More About

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)
```

Version History

Introduced in R2011a

See Also

`getOneWayTurnover` | `setTurnover` | `setInitPort` | `setCosts`

Topics

“Working with One-Way Turnover Constraints Using Portfolio Object” on page 4-84
“Working with One-Way Turnover Constraints Using PortfolioCVaR Object” on page 5-75
“Working with One-Way Turnover Constraints Using PortfolioMAD Object” on page 6-73
“Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
“Portfolio Set for Optimization Using Portfolio Objects” on page 4-8
“Portfolio Set for Optimization Using PortfolioCVaR Object” on page 5-8
“Portfolio Set for Optimization Using PortfolioMAD Object” on page 6-7

setProbabilityLevel

Set probability level for VaR and CVaR calculations

Syntax

```
obj = setProbabilityLevel(obj,ProbabilityLevel)
```

Description

`obj = setProbabilityLevel(obj,ProbabilityLevel)` sets probability level for VaR and CVaR calculations for a `PortfolioCVaR` object. For details on the workflow, see “PortfolioCVaR Object Workflow” on page 5-16.

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
```

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`

Data Types: `object`

`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)
```

Version History

Introduced in R2012b

See Also

`setScenarios`

Topics

"What Are Scenarios?" on page 5-36
"Conditional Value-at-Risk" on page 5-5

External Websites

Getting Started with Portfolio Optimization (4 min 13 sec)
CVaR Portfolio Optimization (4 min 56 sec)

setSolver

Choose main solver and specify associated solver options for portfolio optimization

Syntax

```
obj = setSolver(obj,solverType)
```

```
obj = setSolver(obj,solverType,Name,Value)
```

```
obj = setSolver(obj,solverType,optimoptions)
```

Description

`obj = setSolver(obj,solverType)` selects the main solver and enables you to specify associated solver options for portfolio optimization for `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` objects. For details on the respective workflows when using these different objects, see "Portfolio Object Workflow" on page 4-17, "PortfolioCVaR Object Workflow" on page 5-16, and "PortfolioMAD Object Workflow" on page 6-15.

`obj = setSolver(obj,solverType,Name,Value)` selects the main solver and enables you to specify associated solver options for portfolio optimization for portfolio objects with additional options specified by using one or more `Name,Value` pair arguments.

`obj = setSolver(obj,solverType,optimoptions)` selects the main solver and enables you to specify associated solver options for portfolio optimization for portfolio objects with an `optimoptions` object.

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 `'interior-point'` and to turn off the display.

```
p = PortfolioCVaR;
p = setSolver(p, 'fmincon', 'Algorithm', 'interior-point', 'Display', 'off');
display(p.solverOptions.Algorithm);
```

```
interior-point
```

```
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 `'interior-point'` and to turn off the display.

```
p = PortfolioCVaR;
options = optimoptions('fmincon', 'Algorithm', 'interior-point', 'Display', 'off');
p = setSolver(p, 'fmincon', options);
display(p.solverOptions.Algorithm);
```

```
interior-point
```

```
display(p.solverOptions.Display);
```

```
off
```

Set `'TrustRegionCP'` as the Solver Type with Default Options for a PortfolioCVaR Object

Use `'TrustRegionCP'` as the `solverType` with default options.

```
p = PortfolioCVaR;
p = setSolver(p,'TrustRegionCP');
display(p.solverType);
```

```
trustregioncp
```

```
display(p.solverOptions);
```

```
  struct with fields:

                MaxIterations: 1000
         AbsoluteGapTolerance: 1.0000e-07
         RelativeGapTolerance: 1.0000e-05
       NonlinearScalingFactor: 1000
       ObjectiveScalingFactor: 1000
            MainSolverOptions: [1x1 optim.options.Linprog]
                      Display: 'off'
                CutGeneration: 'basic'
     MaxIterationsInactiveCut: 30
           ActiveCutTolerance: 1.0000e-07
                  ShrinkRatio: 0.7500
    TrustRegionStartIteration: 2
                 InitialDelta: 0.5000
                   DeltaLimit: 1000000
```

Set 'TrustRegionCP' as the Solver Type with 'ShrinkRatio' for a PortfolioCVaR Object

Use the name-value pair `'ShrinkRatio'` to shrink the size of the trust region by `0.75`.

```
p = PortfolioCVaR;
p = setSolver(p,'TrustRegionCP','ShrinkRatio',0.75);
display(p.solverType);
```

```
trustregioncp
```

```
display(p.solverOptions);
```

```
  struct with fields:

                MaxIterations: 1000
         AbsoluteGapTolerance: 1.0000e-07
         RelativeGapTolerance: 1.0000e-05
       NonlinearScalingFactor: 1000
       ObjectiveScalingFactor: 1000
            MainSolverOptions: [1x1 optim.options.Linprog]
                      Display: 'off'
                CutGeneration: 'basic'
     MaxIterationsInactiveCut: 30
           ActiveCutTolerance: 1.0000e-07
                  ShrinkRatio: 0.7500
    TrustRegionStartIteration: 2
                 InitialDelta: 0.5000
                   DeltaLimit: 1000000
```

Set 'TrustRegionCP' as the Solver Type and Change the Main Solver Option for a PortfolioCVaR Object

For the main solver, continue using the `dual-simplex` algorithm with no display, but tighten its termination tolerance to `1e8`.

```
p = PortfolioCVaR;
options = optimoptions('linprog','Algorithm','Dual-Simplex','Display','off','OptimalityTolerance
p = setSolver(p,'TrustRegionCP','MainSolverOptions',options);
display(p.solverType)
```

```
trustregioncp
```

```
display(p.solverOptions)
```

```
  struct with fields:

                MaxIterations: 1000
         AbsoluteGapTolerance: 1.0000e-07
         RelativeGapTolerance: 1.0000e-05
       NonlinearScalingFactor: 1000
       ObjectiveScalingFactor: 1000
            MainSolverOptions: [1x1 optim.options.Linprog]
                      Display: 'off'
                CutGeneration: 'basic'
     MaxIterationsInactiveCut: 30
           ActiveCutTolerance: 1.0000e-07
                  ShrinkRatio: 0.7500
    TrustRegionStartIteration: 2
                 InitialDelta: 0.5000
                   DeltaLimit: 1000000
```

```
display(p.solverOptions.MainSolverOptions.Algorithm)
```

```
dual-simplex
```

```
display(p.solverOptions.MainSolverOptions.Display)
```

```
off
```

```
display(p.solverOptions.MainSolverOptions.TolFun)
```

```
   100000000
```

For the main solver, use the `interior-point` algorithm with no display and with a termination tolerance of `1e7`.

```
p = PortfolioCVaR;
options = optimoptions('linprog','Algorithm','interior-point','Display','off','OptimalityToleran
p = setSolver(p,'TrustRegionCP','MainSolverOptions',options);
display(p.solverType)
```

```
trustregioncp
```

```
display(p.solverOptions)
```

```
  struct with fields:

                MaxIterations: 1000
         AbsoluteGapTolerance: 1.0000e-07
         RelativeGapTolerance: 1.0000e-05
       NonlinearScalingFactor: 1000
       ObjectiveScalingFactor: 1000
            MainSolverOptions: [1x1 optim.options.Linprog]
                      Display: 'off'
                CutGeneration: 'basic'
     MaxIterationsInactiveCut: 30
           ActiveCutTolerance: 1.0000e-07
                  ShrinkRatio: 0.7500
    TrustRegionStartIteration: 2
                 InitialDelta: 0.5000
                   DeltaLimit: 1000000
```

```
display(p.solverOptions.MainSolverOptions.Algorithm)
```

```
interior-point
```

```
display(p.solverOptions.MainSolverOptions.Display)
```

```
off
```

```
display(p.solverOptions.MainSolverOptions.TolFun)
```

```
    10000000
```

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', 'off');
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','on');
p = setSolver(p, 'fmincon', options);
display(p.solverOptions.Algorithm);
```

```
active-set
```

```
display(p.solverOptions.Display);
```

```
off
```

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`

Data Types: `object`

`solverType` — Solver to use for portfolio optimization

character vector | string

Solver to use for portfolio optimization, specified using a character vector or string for the supported `solverType`.

The solverType input argument depends on which type of object (obj) is being used for a portfolio optimization.

For a Portfolio object, the supported solverType are:

- 'lcprog' (Default).
  - The 'lcprog' solver uses linear complementary programming with Lemke's algorithm with control variables name-value pair arguments for 'maxiter', 'tiebreak', 'tolpiv'. For more information about 'lcprog' name-value pair options, see “Portfolio Object Name-Value Pair Arguments” on page 15-1383.
- 'fmincon'
  - The default algorithm for 'fmincon' is 'sqp'. For more information about 'fmincon' name-value pair options, see “Portfolio Object Name-Value Pair Arguments” on page 15-1383.
- 'quadprog'
  - The default algorithm for 'quadprog' is interior-point-convex. For more information about 'quadprog' name-value pair options, see “Portfolio Object Name-Value Pair Arguments” on page 15-1383.

For a PortfolioCVaR object, the supported solverType are:

- 'TrustRegionCP' (Default)
  - 'TrustRegionCP' is an implementation of Kelley's [1] cutting-plane method for convex optimization. For more information about 'TrustRegionCP' name-value pair options, see “Name-Value Pair Arguments for 'TrustRegionCP' and 'ExtendedCP'” on page 15-1385.
- 'ExtendedCP'
  - 'ExtendedCP' is an implementation of Kelley's [1] cutting-plane method for convex optimization. For more information about 'ExtendedCP' name-value pair options, see “Name-Value Pair Arguments for 'TrustRegionCP' and 'ExtendedCP'” on page 15-1385.
- 'fmincon'
  - The default algorithm for 'fmincon' is 'sqp'. For more information about 'fmincon' name-value pair options, see “PortfolioCVaR Object Name-Value Pair Arguments” on page 15-1384.
- 'cuttingplane'
  - The 'cuttingplane' solver is an implementation of Kelley's [1] cutting-plane method for convex optimization with name-value pair arguments for 'MaxIter', 'Abstol', 'Reltol' and 'MainSolverOptions'. For more information about 'cuttingplane' name-value pair options, see “PortfolioCVaR Object Name-Value Pair Arguments” on page 15-1384.

For a PortfolioMAD object, the supported solverType are:

- 'TrustRegionCP' (Default)
  - 'TrustRegionCP' is an implementation of Kelley's [1] cutting-plane method for convex optimization. For more information about 'TrustRegionCP' name-value pair options, see “Name-Value Pair Arguments for 'TrustRegionCP' and 'ExtendedCP'” on page 15-1385.
- 'ExtendedCP'

  - 'ExtendedCP' is an implementation of Kelley's [1] cutting-plane method for convex optimization. For more information about 'ExtendedCP' name-value pair options, see "Name-Value Pair Arguments for 'TrustRegionCP' and 'ExtendedCP'" on page 15-1385.
- 'fmincon'
  - The default algorithm for 'fmincon' is the 'sqp' algorithm and 'GradObj' set to 'on'. For more information about 'fmincon' name-value pair options, see "PortfolioMAD Object Name-Value Pair Arguments" on page 15-1385.

Note setSolver can also configure solver options for 'linprog'. linprog is a helper solver used in estimating efficient frontier problems for a Portfolio, PorfolioCVaR, or PortfolioMAD object. The default algorithm for 'linprog' is 'dual-simplex'. For more information about 'linprog' name-value pair options, see "Name-Value Pair Arguments" on page 15-1383. For more details on using a helper solver, see "Solver Guidelines for Portfolio Objects" on page 4-111, "Solver Guidelines for PortfolioCVaR Objects" on page 5-97, or "Solver Guidelines for PortfolioMAD Objects" on page 6-93.

Data Types: char | string

optimoptions — optimoptions object
object

(Optional) optimoptions object, specified as an optimoptions object that is created using optimoptions from Optimization Toolbox. For example:

```
p = setSolver(p,'fmincon',optimoptions('fmincon','Display','iter'));
```

Note optimoptions is the default and recommended method to set solver options, however optimset is also supported.

Data Types: object

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: p = setSolver(p,'cuttingplane','MainSolverOptions',options) sets cuttingplane options for a PortfolioCVaR object.

Depending on the obj type (Portfolio, PortfolioCVaR, or PortfolioMAD) and the specified solverType, the options for the associated name-value pair arguments are different.

Portfolio Object Name-Value Pair Arguments

- For a Portfolio object using a solverType of lcprog, choose a name-value value in this table.

Value	Description
'maxiter'	Maximum number of iterations,specified as the comma-separated pair consisting of 'MaxIter' and a positive integer. The default value is 1 + n^3, where n is the dimension of the input.
'tiebreak'	Method to break ties for pivot selection, specified as the comma-separated pair consisting of 'tiebreak' and one of the following options: • first - Select pivot with lowest index. • last - Select pivot with highest index. • random - Select a pivot at random. The default value is first.
'tolpiv'	Pivot tolerance below which a number is considered to be zero, specified as the comma-separated pair consisting of 'tolpiv' and a numeric value. The default value is 1.0e-9.

- For a Portfolio object using a solverType of fmincon, see “options” to choose name-value pair arguments.
- For a Portfolio object using a solverType of linprog, see “options” to choose name-value pair arguments.
- For a Portfolio object using a solverType of quadprog, see “options” to choose name-value pair arguments.

PortfolioCVaR Object Name-Value Pair Arguments

- For a PortfolioCVaR object using a solverType of fmincon, see “options” to choose name-value pair arguments.
- For a PorfolioCVaR object using a solverType of 'TrustRegionCP' or 'ExtendedCP', see “Name-Value Pair Arguments for 'TrustRegionCP' and 'ExtendedCP'” on page 15-1385 to choose name-value pair arguments.
- For a PorfolioCVaR object using a solverType of 'cuttingplane', choose a name-value pair value in this table.

Value	Description
'MaxIter'	Maximum number of iterations, specified as the comma-separated pair consisting of 'MaxIter' and a positive integer. The default value is 1000.
'AbsTol'	Absolute stopping tolerance, specified as the comma-separated pair consisting of 'AbsTol' and a positive scalar. The default value is 1e6.
'RelTol'	Relative stopping tolerance, specified as the comma-separated pair consisting of 'RelTol' and a positive scalar. The default value is 1e5.
'MainSolverOptions'	Options for the main solver linprog, specified as the comma-separated pair consisting of 'MainSolverOptions' and an optimoptions object. The default is optimoptions('linprog','Algorithm','Dual-Simplex','Display','off').

- For a `PortfolioCVaR` object using a `solverType` of `linprog`, see “options” to choose name-value pair arguments.

`PortfolioMAD` Object Name-Value Pair Arguments

- For a `PortfolioMAD` object using a `solverType` of `fmincon`, see “options” to choose name-value pair arguments.
- For a `PorfolioMAD` object using a `solverType` of `'TrustRegionCP'` or `'ExtendedCP'`, see “Name-Value Pair Arguments for 'TrustRegionCP' and 'ExtendedCP'” on page 15-1385 to choose name-value pair arguments.
- For a `PortfolioMAD` object using a `solverType` of `linprog`, see “options” to choose name-value pair arguments.

Name-Value Pair Arguments for `'TrustRegionCP'` and `'ExtendedCP'`

For a `PortfolioCVaR` or `PortfolioMAD` object using a `solverType` of `'TrustRegionCP'` or `'ExtendedCP'`, choose a name-value pair value in this table.

Value	Description
`'MaxIterations'`	Maximum number of iterations, specified as the comma-separated pair consisting of `'MaxIterations'` and a positive real number. The default value is `1e3`.
`'NonlinearScalingFactor'`	Scales the nonlinear function and its gradient by a factor, specified as the comma-separated pair consisting of `'NonlinearScalingFactor'` and a positive real number. The default value is `1e3`.
`'ObjectiveScalingFactor'`	Scales the objective function by a factor, specified as the comma-separated pair consisting of `'ObjectiveScalingFactor'` and a positive real number. The default value is `1e3`.
`'AbsoluteGapTolerance'`	Solver stops if the absolute difference between the approximated nonlinear function value and its true value is less than or equal to `AbsoluteGapTolerance`, specified as the comma-separated pair consisting of `'AbsoluteGapTolerance'` and a positive real number. The default value is `1e7`.
`'RelativeGapTolerance'`	Solver stops if the relative difference between the approximated nonlinear function value and its true value is less than or equal to `RelativeGapTolerance`, specified as the comma-separated pair consisting of `'RelativeGapTolerance'` and a positive real number. The default value is `1e5`.
`'Display'`	Level of display, specified as the comma-separated pair consisting of `'Display'` and a supported value of: • `'iter'` displays output at each iteration and gives the technical exit message. • `'final'` displays just the final output and gives the final technical exit message. • `'off'` is the default and displays no output.

Value	Description
'CutGeneration'	Method to add the cut, specified as the comma-separated pair consisting of 'CutGeneration' and a supported value of: • 'basic' is the default and the new cut is added at the latest solution found. • 'midway' is where the new cut is added at the mid point between the latest and previous solution found.
'MaxIterationsInactiveCut'	Removes constraints that are not active for the last MaxIterationsInactiveCut iterations, specified as the comma-separated pair consisting of 'MaxIterationsInactiveCut' and a positive integer. The default value is 30.
'ActiveCutTolerance'	Determines if the cuts are active and is used together with MaxIterationsInactiveCut to decide which cuts to remove from the LP subproblem, specified as the comma-separated pair consisting of 'ActiveCutTolerance' and a real number. The default value is 1e7.
'MainSolverOptions'	Options for the main solver linprog, specified as the comma-separated pair consisting of 'MainSolverOptions' and an optimoptions object. The default is optimoptions('linprog','Algorithm','Dual-Simplex','Display','off').
'TrustRegionStartIteration'	Use this parameter only for a solverType of 'TrustRegionCP'. Solver starts to apply the trust region heuristic at TrustRegionStartIteration. Nonnegative integer. Default is 2.
'ShrinkRatio'	Use this parameter only for a solverType of 'TrustRegionCP'. If the approximated functions are not agreeing well in the previous iterations, the algorithm will shrink the size of trust region by the ShrinkRatio. Nonnegative real between 0 and 1. Default is 0.75.
'InitialDelta'	Use this parameter only for a solverType of 'TrustRegionCP'. Value to initialize trust region. Nonnegative real. Default is 0.5.
'DeltaLimit'	Use this parameter only for a solverType of 'TrustRegionCP'. The trust region of the approximated functions is bounded by DeltaLimit during the iterations. The DeltaLimit value is a nonnegative real and the default value is 1e6. **Note** Modifying 'DeltaLimit' might result in the solver not being able to find a solution. If you modify 'DeltaLimit' without specifying a value for 'InitialDelta', the 'InitialDelta' is automatically set to InitialDelta' = 'DeltaLimit'/2.
'DeltaLowerBound'	Use this parameter only for a solverType of 'TrustRegionCP'. Use 'DeltaLowerBound' to set the lower bound for the trust-region radius. The 'DeltaLowerBound' numeric value must be in [0,1] and you can include 0 and 1. The default value is 0.01.

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,Name,Value);
```

Algorithms

To solve the efficient frontier of a portfolio, one version of the portfolio optimization problem minimizes the portfolio risk `Risk`(x), subject to a target return, and other linear constraints specified for the `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object. For the definition of portfolio risk and return, see “Risk Proxy” on page 4-5 and “Return Proxy” on page 4-4.

$$Minimize_x Risk(x)$$

$$\text{subject to } \; Return(x) \geq TargetReturn$$

$$Ax \leq b$$

$$A_{eq}x = b_{eq}$$

$$lb \leq x \leq ub$$

An alternative version of the portfolio optimization problem maximizes the expected return of the portfolio, subject to a target risk and other linear constraints specified for the `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object.

$$Maximize_x Return(x)$$

$$\text{subject to } \; Risk(x) \leq \text{Targ}etRisk$$

$$Ax \leq b$$

$$A_{eq}x = b_{eq}$$

$$lb \leq x \leq ub$$

The return proxy is always a linear function. Therefore, depending on the risk proxy and whether it is used as the objective or constraints, the problem needs to be solved by different solvers. For example, `quadprog` is appropriate for problems with a quadratic function as the objective and only linear constraints, and `fmincon` is appropriate for problems with nonlinear objective or constraints. In addition, there are solvers in Financial Toolbox suitable for certain special types of problems, such as the `solverType` `lcprog`, `'TrustRegionCP'`, or `'ExtendedCP'`.

Version History

Introduced in R2011a

R2023a: Added name-value argument `'DeltaLowerBound'` for `solverType` of `'TrustRegionCP'`

When using a `solverType` of `'TrustRegionCP'`, you can use the name-value argument `'DeltaLowerBound'` to set the lower bound for the trust-region radius.

R2023a: Renamed `'MasterSolverOptions'` name-value argument to `'MainSolverOptions'`

The `'MasterSolverOptions'` name-value argument is renamed to `'MainSolverOptions'`. The use of `'MasterSolverOptions'` name-value argument is discouraged.

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` | `setTurnover` | `setInitPort` | `setCosts` | `setSolverMINLP`

Topics

“Working with One-Way Turnover Constraints Using Portfolio Object” on page 4-84
“Working with One-Way Turnover Constraints Using PortfolioCVaR Object” on page 5-75
“Working with One-Way Turnover Constraints Using PortfolioMAD Object” on page 6-73
“Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
“Portfolio Set for Optimization Using Portfolio Objects” on page 4-8
“Portfolio Set for Optimization Using PortfolioCVaR Object” on page 5-8
“Portfolio Set for Optimization Using PortfolioMAD Object” on page 6-7
“Choosing and Controlling the Solver for Mean-Variance Portfolio Optimization” on page 4-110
“Choosing and Controlling the Solver for PortfolioCVaR Optimizations” on page 5-95
“Choosing and Controlling the Solver for PortfolioMAD Optimizations” on page 6-91

setSolverMINLP

Choose mixed integer nonlinear programming (MINLP) solver for portfolio optimization

Syntax

```
obj = setSolverMINLP(obj,solverTypeMINLP)
```

```
obj = setSolverMINLP(___,Name,Value)
```

Description

`obj = setSolverMINLP(obj,solverTypeMINLP)` selects the mixed integer nonlinear programming (MINLP) solver and enables you to specify associated solver options for portfolio optimization for a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object.

When any one or any combination of `'Conditional'` `BoundType`, `MinNumAssets`, or `MaxNumAssets` constraints are active, the portfolio problem is formulated by adding `NumAssets` binary variables. The binary variable `0` indicates that an asset is not invested and the binary variable `1` indicates that an asset is invested. For more information on using `'Conditional'` `BoundType`, see `setBounds`. For more information on specifying `MinNumAssets` and `MaxNumAssets`, see `setMinMaxNumAssets`.

If you use the `estimate` functions with a `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` object for which any of the `'Conditional'` `BoundType`, `MinNumAssets`, or `MaxNumAssets` constraints are active, MINLP solver is automatically used. For details on MINLP, see “Algorithms” on page 15-1399.

`obj = setSolverMINLP(___,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Configure MINLP Solver and Options When Using a Portfolio Object

Configure the MINLP solver for a three-asset portfolio for which you have the mean and covariance values of the asset returns.

```
AssetMean = [ 0.0101110; 0.0043532; 0.0137058 ];
AssetCovar = [ 0.00324625 0.00022983 0.00420395;
               0.00022983 0.00049937 0.00019247;
               0.00420395 0.00019247 0.00764097 ];
p = Portfolio('AssetMean', AssetMean, 'AssetCovar', AssetCovar);
```

When working with a `Portfolio` object, use `setBounds` with a `'Conditional'BoundType` constraint to set xi = `0` or `0.02` <= xi <= `0.5` for all i = 1,...`NumAssets`.

```
p = setBounds(p, 0.02, 0.5,'BoundType', 'Conditional', 'NumAssets', 3);
```

When working with a `Portfolio` object, use `setMinMaxNumAssets` function to set up `MinNumAssets` and `MaxNumAssets` constraints for a portfolio. This sets limit constraints for the `Portfolio` object, where the total number of allocated assets satisfying the constraints is between

`MinNumAssets` and `MaxNumAssets`. Setting `MinNumAssets` = `MaxNumAssets` = 2, specifies that only two of the three assets are invested in the portfolio.

```
p = setMinMaxNumAssets(p, 2, 2);
```

Three different MINLP solvers (`OuterApproximation`, `ExtendedCP`, `TrustRegionCP`) use the cutting plane method. Use the `setSolverMINLP` function to configure the `OuterApproximation` solver and options.

```
pint = setSolverMINLP(p,'OuterApproximation', 'NonlinearScalingFactor', 1e4, 'Display', 'iter',
pint.solverTypeMINLP
```

```
ans = 
'OuterApproximation'
```

```
pint.solverOptionsMINLP
```

```
ans = struct with fields:
                           MaxIterations: 1000
                    AbsoluteGapTolerance: 1.0000e-07
                    RelativeGapTolerance: 1.0000e-05
                  NonlinearScalingFactor: 10000
                  ObjectiveScalingFactor: 1000
                                 Display: 'iter'
                           CutGeneration: 'basic'
                MaxIterationsInactiveCut: 30
                      ActiveCutTolerance: 1.0000e-07
                    IntMainSolverOptions: [1x1 optim.options.Intlinprog]
    NumIterationsEarlyIntegerConvergence: 30
```

You can also configure the options for `intlinprog`,which is the Main solver for mixed integer linear programming problems in the MINLP solver.

```
pint = setSolverMINLP(p,'OuterApproximation', 'IntMainSolverOptions', optimoptions('intlinprog',
pint.solverOptionsMINLP.IntMainSolverOptions
```

```
ans = 
  intlinprog options:

   Set properties:
                          Display: 'off'

   Default properties:
             AbsoluteGapTolerance: 0
                       BranchRule: 'reliability'
              ConstraintTolerance: 1.0000e-04
                    CutGeneration: 'basic'
                 CutMaxIterations: 10
                       Heuristics: 'basic'
               HeuristicsMaxNodes: 50
                IntegerPreprocess: 'basic'
                 IntegerTolerance: 1.0000e-05
                  LPMaxIterations: 'max(30000,10*(numberOfEqualities+numberOfInequalities+number
            LPOptimalityTolerance: 1.0000e-07
                MaxFeasiblePoints: Inf
                         MaxNodes: 10000000
                          MaxTime: 7200
                    NodeSelection: 'simplebestproj'
                 ObjectiveCutOff: Inf
    ObjectiveImprovementThreshold: 0
                       OutputFcn: []
                         PlotFcn: []
             RelativeGapTolerance: 1.0000e-04
                  RootLPAlgorithm: 'dual-simplex'
              RootLPMaxIterations: 'max(30000,10*(numberOfEqualities+numberOfInequalities+number(
```

Configure MINLP Solver and Options When Using a PortfolioCVaR Object

Configure the MINLP solver for a 12 asset portfolio that is using semicontinuous and cardinality constraints.

```
load CAPMuniverse
p = PortfolioCVaR('AssetList',Assets(1:12));
p = simulateNormalScenariosByData(p, Data(:,1:12),20000,'missingdata',true);
p = setProbabilityLevel(p, 0.95);
```

When working with a `PortfolioCVaR` object, the `setMinMaxNumAssets` function enables you to set up the limits on the number of assets invested. The following example specifies that a minimum of five assets and a maximum of 10 assets should be invested using `setMinMaxNumAssets` and the investments should be greater than 4% and less than 45% using `setBounds`.

```
p = setMinMaxNumAssets(p, 5, 10);
p = setBounds(p, 0.04, 0.45, 'BoundType','conditional');
```

Three different MINLP solvers (`OuterApproximation`, `ExtendedCP`, `TrustRegionCP`) use the cutting plane method. Use the `setSolverMINLP` function to configure the `OuterApproximation` solver and options.

```
pint = setSolverMINLP(p,'OuterApproximation','NonlinearScalingFactor', 1e4, 'Display', 'iter', '(
pint.solverTypeMINLP
```

```
ans = 
'OuterApproximation'
```

```
pint.solverOptionsMINLP
```

```
ans = struct with fields:
                       MaxIterations: 1000
                si AbsoluteGapTolerance: 1.0000e-07
```

```
pint = setSolverMINLP(p,'OuterApproximation','IntMainSolverOptions', optimoptions('intlinprog','D
pint.solverOptionsMINLP.IntMainSolverOptions

ans = 
  intlinprog options:

   Set properties:
                           Display: 'off'

   Default properties:
             AbsoluteGapTolerance: 0
                       BranchRule: 'reliability'
              ConstraintTolerance: 1.0000e-04
                    CutGeneration: 'basic'
                 CutMaxIterations: 10
                       Heuristics: 'basic'
               HeuristicsMaxNodes: 50
                IntegerPreprocess: 'basic'
                 IntegerTolerance: 1.0000e-05
                  LPMaxIterations: 'max(30000,10*(numberOfEqualities+numberOfInequalities+number
            LPOptimalityTolerance: 1.0000e-07
                MaxFeasiblePoints: Inf
                         MaxNodes: 10000000
                          MaxTime: 7200
                    NodeSelection: 'simplebestproj'
                  ObjectiveCutOff: Inf
    ObjectiveImprovementThreshold: 0
                        OutputFcn: []
                          PlotFcn: []
             RelativeGapTolerance: 1.0000e-04
                  RootLPAlgorithm: 'dual-simplex'
              RootLPMaxIterations: 'max(30000,10*(numberOfEqualities+numberOfInequalities+number
```

Configure MINLP Solver and Options When Using a PortfolioMAD Object

Configure the MINLP solver for a 12 asset portfolio that is using semicontinuous and cardinality constraints.

```
load CAPMuniverse
p = PortfolioMAD('AssetList',Assets(1:12));
p = simulateNormalScenariosByData(p, Data(:,1:12), 20000 ,'missingdata',true);
```

When working with a `PortfolioMAD` object, the `setMinMaxNumAssets` function enables you to set up the limits on the number of assets invested. The following example specifies that a minimum of five assets and a maximum of 10 assets should be invested using `setMinMaxNumAssets` and the investments should be greater than 4% and less than 45% using `setBounds`.

```
p = setMinMaxNumAssets(p, 5, 10);
p = setBounds(p, 0.04, 0.45, 'BoundType', 'conditional');
```

Three different MINLP solvers (`OuterApproximation`, `ExtendedCP`, `TrustRegionCP`) use the cutting plane method. Use the `setSolverMINLP` function to configure the `OuterApproximation` solver and options.

```
pint = setSolverMINLP(p,'OuterApproximation', 'NonlinearScalingFactor', 1e4, 'Display', 'iter',
pint.solverTypeMINLP
```

```
ans = 
'OuterApproximation'
```

```
pint.solverOptionsMINLP
```

```
ans = struct with fields:
                           MaxIterations: 1000
                    AbsoluteGapTolerance: 1.0000e-07
                    RelativeGapTolerance: 1.0000e-05
                  NonlinearScalingFactor: 10000
                  ObjectiveScalingFactor: 1000
                                 Display: 'iter'
                           CutGeneration: 'basic'
                MaxIterationsInactiveCut: 30
                      ActiveCutTolerance: 1.0000e-07
                    IntMainSolverOptions: [1x1 optim.options.Intlinprog]
    NumIterationsEarlyIntegerConvergence: 30
```

You can also configure the options for `intlinprog`,which is the Main solver for mixed integer linear programming problems in the MINLP solver.

```
pint = setSolverMINLP(p,'OuterApproximation', 'IntMainSolverOptions', optimoptions('intlinprog',
pint.solverOptionsMINLP.IntMainSolverOptions
```

```
ans = 
  intlinprog options:

   Set properties:
                          Display: 'off'

   Default properties:
             AbsoluteGapTolerance: 0
                       BranchRule: 'reliability'
              ConstraintTolerance: 1.0000e-04
                    CutGeneration: 'basic'
                 CutMaxIterations: 10
                       Heuristics: 'basic'
               HeuristicsMaxNodes: 50
                IntegerPreprocess: 'basic'
                 IntegerTolerance: 1.0000e-05
                  LPMaxIterations: 'max(30000,10*(numberOfEqualities+numberOfInequalities+number
             LPOptimalityTolerance: 1.0000e-07
                MaxFeasiblePoints: Inf
                         MaxNodes: 10000000
                          MaxTime: 7200
                    NodeSelection: 'simplebestproj'
                  ObjectiveCutOff: Inf
    ObjectiveImprovementThreshold: 0
                        OutputFcn: []
                          PlotFcn: []
             RelativeGapTolerance: 1.0000e-04
                  RootLPAlgorithm: 'dual-simplex'
              RootLPMaxIterations: 'max(30000,10*(numberOfEqualities+numberOfInequalities+number
```

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

Data Types: object

solverTypeMINLP — MINLP solver for portfolio optimization
character vector with a value of 'OuterApproximation', 'ExtendedCP', or 'TrustRegionCP' | string with a value of "OuterApproximation", "ExtendedCP", or "TrustRegionCP"

MINLP solver for portfolio optimization when any one or any combination of 'Conditional' BoundType, MinNumAssets, or MaxNumAssets constraints are active. Specify solverTypeMINLP using a character vector or string with a value of 'OuterApproximation', 'ExtendedCP', or 'TrustRegionCP'.

For a Portfolio object, the default value of the solverTypeMINLP is 'OuterApproximation' with the following default settings for name-value pairs for setSolverMINLP:

- MaxIterations — 1000
- AbsoluteGapTolerance — 1.0000e-07
- RelativeGapTolerance — 1.0000e-05
- Display — 'off'
- NonlinearScalingFactor — 1000
- ObjectiveScalingFactor — 1000
- CutGeneration — 'basic'
- MaxIterationsInactiveCut — 30
- NumIterationsEarlyIntegerConvergence — 30
- ActiveCutTolerance — 1.0000e-07
- IntMainSolverOptions — optimoptions('intlinprog','Algorithm','Dual-Simplex','Display','off')

For a PortfolioCVaR and PortfolioMAD object, the default value of the solverTypeMINLP is 'TrustRegionCP' with the following default settings for name-value pairs for setSolverMINLP:

- MaxIterations — 1000
- AbsoluteGapTolerance — 1.0000e-07
- RelativeGapTolerance — 1.0000e-05
- Display — 'off'
- NonlinearScalingFactor — 1000
- ObjectiveScalingFactor — 1000

- CutGeneration — 'basic'
- MaxIterationsInactiveCut — 30
- NumIterationsEarlyIntegerConvergence — 30
- ActiveCutTolerance — 1.0000e-07
- TrustRegionStartIteration - 2
- ShrinkRatio — 0.75
- InitialDelta — 0.5
- DeltaLimit — 1e6
- DeltaLowerBound — 0.01
- IntMainSolverOptions — optimoptions('intlinprog','Algorithm','Dual-Simplex','Display','off')

Data Types: char | string

Name-Value Pair Arguments or optimoptions Object

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: p = setSolverMINLP(p,'ExtendedCP','MaxIterations',10000,'NonlinearScalingFactor',1000)

MaxIterations — Maximum number of iterations
1000 (default) | nonnegative integer

Maximum number of iterations, specified as the comma-separated pair consisting of 'MaxIterations' and a nonnegative integer value.

Data Types: double

NonlinearScalingFactor — Scaling factor for nonlinear function and gradient
1000 (default) | nonnegative real

Scaling factor for nonlinear function and gradient, specified as the comma-separated pair consisting of 'NonlinearScalingFactor' and a nonnegative real value.

Data Types: double

ObjectiveScalingFactor — Scales the objective function used by MainSolver by a factor
1000 (default) | nonnegative real

Scales the objective function used by MainSolver by a factor, specified as the comma-separated pair consisting of 'ObjectiveScalingFactor' and a nonnegative real value.

Data Types: double

AbsoluteGapTolerance — Solver stops if absolute difference between approximated nonlinear function value and its true value is less than or equal to AbsoluteGapTolerance
0.0000001 (default) | nonnegative real

The solver stops if the absolute difference between the approximated nonlinear function value and its true value is less than or equal to AbsoluteGapTolerance. AbsoluteGapTolerance is specified as the comma-separated pair consisting of 'AbsoluteGapTolerance' and a nonnegative real value.

Data Types: double

RelativeGapTolerance — Solver stops if the relative difference between approximated nonlinear function value and its true value is less than or equal to RelativeGapTolerance
0.00001 (default) | nonnegative real

The solver stops if the relative difference between the approximated nonlinear function value and its true value is less than or equal to RelativeGapTolerance. RelativeGapTolerance is specified as the comma-separated pair consisting of 'AbsoluteGapTolerance' and a nonnegative real value.

Data Types: double

Display — Display output format
'off' (default) | character vector with value of 'iter', 'final', or 'off'

Display output format, specified as the comma-separated pair consisting of 'Display' and a character vector with a value of:

- 'off' - Display no output
- 'iter' - Display output at each iteration and the technical exit message
- 'final' - Display only the final output and the final technical exit message

Data Types: char

CutGeneration — Cut specification
'basic' (default) | character vector with value of 'midway' or 'basic'

Cut specification, specified as the comma-separated pair consisting of 'CutGeneration' and a character vector with one of these values:

- 'midway' - Add the new cut at the midpoint between the latest and previous solutions found.
- 'basic' - Add the new cut at the latest solution found.

Data Types: char

MaxIterationsInactiveCut — Removes constraints that are not active for last MaxIterationsInactiveCut iterations
30 (default) | nonnegative integer

Removes constraints that are not active for the last MaxIterationsInactiveCut iterations, specified as the comma-separated pair consisting of 'MaxIterationsInactiveCut' and a nonnegative integer value. Generally, the MaxIterationsInactiveCut value is larger than 10.

Data Types: double

NumIterationsEarlyIntegerConvergence — When integer variable solution is stable for the last NumIterationsEarlyIntegerConvergence iterations, the solver computes a final NLP by using latest integer variable solution in the MILP
30 (default) | nonnegative integer

When the integer variable solution is stable for the last NumIterationsEarlyIntegerConvergence iterations, the solver computes a final NLP by using the latest integer variable solution in the MILP. NumIterationsEarlyIntegerConvergence is specified as the comma-separated pair consisting of 'NumIterationsEarlyIntegerConvergence' and a nonnegative integer value.

Data Types: double

ActiveCutTolerance — Determines if the cuts are active
0.0000001 (default) | nonnegative real

Determines if the cuts are active, specified as the comma-separated pair consisting of 'ActiveCutTolerance' and a nonnegative real value. ActiveCutTolerance is used together with MaxIterationsInactiveCut to decide which cuts to remove from the MILP subproblem.

Data Types: double

TrustRegionStartIteration — Solver starts to apply trust region heuristic at TrustRegionStartIteration
2 (default) | nonnegative integer

Solver starts to apply the trust region heuristic at TrustRegionStartIteration, specified as the comma-separated pair consisting of 'TrustRegionStartIteration' and a nonnegative integer.

Note The TrustRegionStartIteration name-value pair argument can only be used with a solverTypeMINLP of 'TrustRegionCP'.

Data Types: double

ShrinkRatio — Ratio to shrink size of trust region
0.75 (default) | nonnegative real between 0 and 1

Ratio to shrink size of trust region, specified as the comma-separated pair consisting of 'ShrinkRatio' and a nonnegative real value between 0 and 1. If the approximated functions do not have good agreement in the previous iterations, the algorithm uses this ratio to shrink the trust-region size.

Note The ShrinkRatio name-value pair argument can only be used with a solverTypeMINLP of 'TrustRegionCP'.

Data Types: double

InitialDelta — Value to initialize trust region
0.5 (default) | nonnegative real

Value to initialize trust region, specified as the comma-separated pair consisting of 'InitialDelta' and a nonnegative real value.

Note The InitialDelta name-value pair argument can only be used with a solverTypeMINLP of 'TrustRegionCP'.

Data Types: double

DeltaLimit — Trust region of the approximated functions is bounded by DeltaLimit during iterations

1e6 (default) | nonnegative real

Trust region of the approximated functions is bounded by DeltaLimit during the iterations, specified as the comma-separated pair consisting of 'DeltaLimit' and a nonnegative real value.

Note The DeltaLimit name-value pair argument can only be used with a solverTypeMINLP of 'TrustRegionCP'.

Modifying 'DeltaLimit' might result in the solver not being able to find a solution.

If you modify 'DeltaLimit' without specifying a value for 'InitialDelta', the 'InitialDelta' is automatically set to InitialDelta' = 'DeltaLimit'/2.

Data Types: double

DeltaLowerBound — Set lower bound for trust-region radius

0.01 (default) | numeric value in [0,1]

Set lower bound for the trust-region radius, specified as the comma-separated pair consisting of 'DeltaLowerBound' and a numeric value in [0,1] that can include 0 and 1.

Note The DeltaLowerBound name-value pair argument can only be used with a solverTypeMINLP of 'TrustRegionCP'.

Data Types: double

IntMainSolverOptions — Options for main solver

optimoptions('intlinprog','Display','off', 'ConstraintTolerance',1e5,'MaxTime',1000,'IPPreprocess','none','CutGeneration','advanced','Heuristics','rins','IntegerTolerance',1e5, 'NodeSelection','mininfeas','LPPreprocess','none') (default) | optimoptions object

Options for the main solverintlinprog, specified as the comma-separated pair consisting of 'IntMainSolverOptions' and an optimoptions object.

Example: 'IntMainSolverOptions', optimoptions('intlinprog','Display','off')

Output Arguments

obj — Updated portfolio object

object

Updated portfolio object, returned as a Portfolio, PortfolioCVaR, or PortfolioMAD object.

More About

MINLP Solvers

All three MINLP solvers (`'OuterApproximation'`, `'ExtendedCP'`, and `'TrustRegionCP'`) defined by `solverTypeMINLP` rely on the cutting plane concept.

These MINLP solvers approximate the nonlinear convex function $f(x)$ by a piecewise linear approximation, which is a sequence of linear cuts around the original function. In this way, the original MINLP is reduced to a sequence of MILP subproblems, each one with a more refined approximation to $f(x)$ than previous MILPs, and yields a more optimal solution. The process continues until the solution found from MILP converges to the true function value within a certain tolerance.

- The `'ExtendedCP'` solver iteratively adds a linear cut at the latest solution found to approximate $f(x)$.
- The `'OuterApproximation'` solver is similar to `'ExtendedCP'`, but they differ in where to add the cut. Instead of using the solution from the latest MILP, `OuterApproximation` uses the values of integer variables from the latest MILP solution and fixes them to reduce the MINLP to a nonlinear programming (NLP) problem. The cut is added at the solution from this NLP problem.
- The `'TrustRegionCP'` solver is a version of `'ExtendedCP'` that is modified to speed up the optimization process. In general, the trust region method uses a model to approximate the true function within a region at each iteration. In the context of the MINLP solver, the model is the maximum of all the added cuts. The true function is the nonlinear function $f(x)$ in the optimization problem. The region of the model is updated based on how well the model approximates the true function for the iteration. This approximation is the comparison of the predicted reduction of the objective function using the model vs. the true reduction.

Tips

You can also use dot notation to specify associated name-value pair options.

```
obj = obj.setSolverMINLP(Name,Value);
```

Note The `solverTypeMINLP` and `solverOptionsMINLP` properties cannot be set using dot notation because they are hidden properties. To set the `solverTypeMINLP` and `solverOptionsMINLP` properties, use the `setSolverMINLP` function directly.

Algorithms

When any one, or any combination of `'Conditional'` `BoundType`, `MinNumAssets`, or `MaxNumAssets` constraints is active, the portfolio problem is formulated by adding `NumAssets` binary variables. The binary variable `0` indicates that an asset is not invested and the binary variable `1` indicates that an asset is invested.

The `MinNumAssets` and `MaxNumAssets` constraints narrow down the number of active positions in a portfolio to the range of [*minN*, *maxN*]. In addition, the `'Conditional'` `BoundType` constraint is to set a lower and upper bound so that the position is either `0` or lies in the range [*minWgt*, *maxWgt*]. These two types of constraints are incorporated into the portfolio optimization model by introducing n variables, ν_i, which only take binary values `0` and `1` to indicate whether the corresponding asset is invested (`1`) or not invested (`0`). Here n is the total number of assets and the constraints can be formulated as the following linear inequality constraints:

$$minN \leq \sum_{i=1}^{n} \upsilon_i \leq \ maxN$$

$$minWgt * \upsilon_i \leq x_i \leq maxWgt * \upsilon_i$$

$$0 \leq \upsilon \leq 1$$

υ_i are integers

In this equation, *minN* and *maxN* are representations for `MinNumAsset` and `MaxNumAsset` that are set using `setMinMaxNumAssets`. Also, *minWgt* and *maxWgt* are representations for `LowerBound` and `UpperBound` that are set using `setBounds`.

The portfolio optimization problem to minimize the variance of the portfolio, subject to achieving a target expected return and some additional linear constraints on the portfolio weights, is formulated as

$$minimize_x \ \ x^T Hx$$

$$s.t. \ \ m^T x \geq TargetReturn$$

$$Ax \leq b$$

$$A_{eq}x = b_{eq}$$

$$lb \leq x \leq ub$$

In this equation, *H* represents the covariance and *m* represents the asset returns.

The portfolio optimization problem to maximize the return, subject to an upper limit on the variance of the portfolio return and some additional linear constraints on the portfolio weights, is formulated as

$$maximize_x \ \ m^T x$$

$$s.t. \ \ x^T Hx \leq TargetRisk$$

$$Ax \leq b$$

$$A_{eq}x = b_{eq}$$

$$lb \leq x \leq ub$$

When the `'Conditional'` `BoundType`, `MinNumAssets`, and `MaxNumAssets` constraints are added to the two optimization problems, the problems become:

$$minimize_{x\upsilon} \ \ x^T Hx$$

$$s.t. \ \ m^T x \geq TargetReturn$$

$$A'[x;\upsilon] \leq b'$$

$$A_{eq}x = b_{eq}$$

$$minN \leq \sum_{i=1}^{n} \upsilon_i \leq maxN$$

$$minWgt(\upsilon_i) \leq x_i \leq maxWgt(\upsilon_i)$$

$$lb \leq x \leq ub$$

$$0 \leq \upsilon \leq 1$$

υ_i are integers

$$
\begin{aligned}
&maximize_{x\upsilon} \;\; m^T x \\
&s.t. \;\; x^T H x \geq TargetRisk \\
&A'[x; \upsilon] \leq b' \\
&A_{eq} x = b_{eq} \\
&minN \leq \sum_{i=1}^{n} \upsilon_i \leq maxN \\
&minWgt * \upsilon_i \leq x_i \leq maxWgt(\upsilon_i) \\
&0 \leq \upsilon \leq 1 \\
&\upsilon_i \text{ are integers}
\end{aligned}
$$

Version History

Introduced in R2018b

R2023a: Added name-value argument `'DeltaLowerBound'` for `solverTypeMINLP` of `'TrustRegionCP'`

When using a `solverTypeMINLP` of `'TrustRegionCP'`, you can use the `'DeltaLowerBound'` name-value argument to set the lower bound for the trust-region radius.

R2023a: Renamed `'IntMasterSolverOptions'` name-value argument to `'IntMainSolverOptions'`

The `'IntMasterSolverOptions'` name-value argument is renamed to `'IntMainSolverOptions'`. The use of `'IntMasterSolverOptions'` name-value argument is discouraged.

References

[1] Bonami, P., Kilinc, M., and J. Linderoth. "Algorithms and Software for Convex Mixed Integer Nonlinear Programs." Technical Report #1664. Computer Sciences Department, University of Wisconsin-Madison, 2009.

[2] Kelley, J. E. "The Cutting-Plane Method for Solving Convex Programs." *Journal of the Society for Industrial and Applied Mathematics.* Vol. 8, Number 4, 1960, pp. 703–712.

[3] Linderoth, J. and S. Wright. "Decomposition Algorithms for Stochastic Programming on a Computational Grid." *Computational Optimization and Applications.* Vol. 24, Issue 2–3, 2003, pp. 207–250.

[4] Nocedal, J., and S. Wright. *Numerical Optimization.* New York: Springer-Verlag, 1999.

See Also

`setBounds` | `setMinMaxNumAssets` | `estimateFrontier` | `estimateFrontierByReturn` | `estimateFrontierByRisk` | `estimateFrontierLimits` | `estimateMaxSharpeRatio` | `setSolver`

Topics

“Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-183
“Mixed-Integer Quadratic Programming Portfolio Optimization: Problem-Based”
“Portfolio Set for Optimization Using Portfolio Objects” on page 4-8
“Choosing and Controlling the Solver for Mean-Variance Portfolio Optimization” on page 4-110
“Choosing and Controlling the Solver for PortfolioCVaR Optimizations” on page 5-95
“Choosing and Controlling the Solver for PortfolioMAD Optimizations” on page 6-91

setScenarios

Set asset returns scenarios by direct matrix

Note Using a `fints` object for the `AssetScenarios` argument of `setScenarios` is not recommended. Use `timetable` instead for financial time series. For more information, see “Convert Financial Time Series Objects (fints) to Timetables”.

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. For details on the workflows, see “PortfolioCVaR Object Workflow” on page 5-16, and “PortfolioMAD Object Workflow” on page 6-15.

`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: []
        MinNumAssets: []
        MaxNumAssets: []
           BoundType: [4x1 categorical]
```

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: []
    MinNumAssets: []
    MaxNumAssets: []
       BoundType: [4x1 categorical]
```

Set Asset Returns Scenarios for a PortfolioCVaR Object Using Timetable Data

To illustrate using the `setScenarios` function with `AssetScenarios` data continued in a `timetable` object, use the `CAPMuniverse.mat` which contains a `timetable` object (`AssetTimeTable`) for returns data.

```
load CAPMuniverse;
AssetsTimeTable.Properties;
head(AssetsTimeTable,5)
```

```
       Time          AAPL         AMZN         CSCO         DELL         EBAY       GOOG       HP
    ___________    _________    _________    _________    _________    _________    ____    _____

    03-Jan-2000     0.088805       0.1742     0.008775    -0.002353      0.12829    NaN       0.0
    04-Jan-2000    -0.084331     -0.08324     -0.05608     -0.08353    -0.093805    NaN     -0.07
    05-Jan-2000     0.014634     -0.14877    -0.003039     0.070984     0.066875    NaN     -0.00
    06-Jan-2000    -0.086538    -0.060072    -0.016619    -0.038847    -0.012302    NaN     -0.06
    07-Jan-2000     0.047368     0.061013       0.0587    -0.037708    -0.000964    NaN      0.02
```

`setScenarios` 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'`).

```
r = PortfolioCVaR;
r = setScenarios(r,AssetsTimeTable,'dataformat','returns');
```

In addition, the `setScenarios` function also extracts asset names or identifiers from a timetable object when the name-value argument `'GetAssetList'` set to `true` (its default value is `false`). If the `'GetAssetList'` value is `true`, the timetable column identifiers are used to set the `AssetList` property of the PortfolioCVaR object. To show this, the formation of the PortfolioCVaR object `r` is repeated with the `'GetAssetList'` flag set to `true`.

```
r = setScenarios(r,AssetsTimeTable,'GetAssetList',true);
disp(r.AssetList')
```

```
    {'AAPL'  }
    {'AMZN'  }
    {'CSCO'  }
    {'DELL'  }
    {'EBAY'  }
    {'GOOG'  }
    {'HPQ'   }
    {'IBM'   }
    {'INTC'  }
    {'MSFT'  }
    {'ORCL'  }
    {'YHOO'  }
    {'MARKET'}
    {'CASH'  }
```

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`

Data Types: `object`

`AssetScenarios` — Scenarios for asset returns or prices

matrix | table | timetable

Scenarios for asset returns or prices, specified as a matrix, `table`, or `timetable` that contains asset data that can be converted into asset returns ([`NumSamples`-by-`NumAssets`] matrix).

`AssetReturns` data can be:

- `NumSamples`-by-`NumAssets` matrix.
- Table of `NumSamples` prices or returns at a given periodicity for an underlying single-period investment horizon for a collection of `NumAssets` assets
- Timetable object with `NumSamples` observations and `NumAssets` time series

If the input data are prices, they can be converted into returns with the `DataFormat` name-value pair 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` | `table` | `timetable`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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 as the comma-separated pair consisting of `'DataFormat'` and 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 the comma-separated pair consisting of `'GetAssetList'` and 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 table or timetable.

If a `table` or `timetable` is passed into this function as `AssetScenarios` and the `GetAssetList` flag is `true`, the column names from the `table` or `timetable` 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);
```

Version History

Introduced in R2012b

See Also

`getScenarios`

Topics

"Asset Returns and Scenarios Using PortfolioCVaR Object" on page 5-36
"Asset Returns and Scenarios Using PortfolioMAD Object" on page 6-34

External Websites

Getting Started with Portfolio Optimization (4 min 13 sec)
CVaR Portfolio Optimization (4 min 56 sec)

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 for a `Portfolio` object. For details on the workflow when using a Portfolio object, see "Portfolio Object Workflow" on page 4-17.

`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: []
       MinNumAssets: []
       MaxNumAssets: []
          BoundType: []
```

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
```

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`.

Data Types: `object`

`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 4-14.

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, [ ]);
```

Version History
Introduced in R2015b

See Also
setTrackingPort | Portfolio

Topics
“Working with Tracking Error Constraints Using Portfolio Object” on page 4-87

“Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
“Tracking Error Constraints” on page 4-14
“Setting Up a Tracking Portfolio” on page 4-39

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 for a `Portfolio` object. For details on the workflow when using a Portfolio object, see “Portfolio Object Workflow” on page 4-17.

`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: []
       MinNumAssets: []
       MaxNumAssets: []
          BoundType: []
```

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
```

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`.

Data Types: `object`

`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, [ ]);
```

Version History

Introduced in R2015b

See Also

`setTrackingError` | `Portfolio`

Topics

“Working with Tracking Error Constraints Using Portfolio Object” on page 4-87
“Portfolio Optimization Examples Using Financial Toolbox™” on page 4-152
“Tracking Error Constraints” on page 4-14
“Setting Up a Tracking Portfolio” on page 4-39

setTurnover

Set up maximum portfolio turnover constraint

Syntax

```
obj = setTurnover(obj,Turnover)
obj = setTurnover(obj,Turnover,InitPort,NumAssets)
```

Description

`obj = setTurnover(obj,Turnover)` sets up maximum portfolio turnover constraint for `Portfolio`, `PortfolioCVaR`, or `PortfolioMAD` objects. For details on the respective workflows when using these different objects, see "Portfolio Object Workflow" on page 4-17, "PortfolioCVaR Object Workflow" on page 5-16, and "PortfolioMAD Object Workflow" on page 6-15.

`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
```

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`

Data Types: `object`

`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);
```

Version History

Introduced in R2011a

See Also

`getOneWayTurnover` | `setOneWayTurnover` | `setInitPort`

Topics

"Working with Average Turnover Constraints Using Portfolio Object" on page 4-81
"Working with Average Turnover Constraints Using PortfolioCVaR Object" on page 5-72
"Portfolio Optimization Examples Using Financial Toolbox™" on page 4-152
"Portfolio Analysis with Turnover Constraints" on page 4-204
"Portfolio Set for Optimization Using Portfolio Objects" on page 4-8
"Portfolio Set for Optimization Using PortfolioCVaR Object" on page 5-8
"Portfolio Set for Optimization Using PortfolioMAD Object" on page 6-7

simulateNormalScenariosByData

Simulate multivariate normal asset return scenarios from data

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. For details on the workflows, see "PortfolioCVaR Object Workflow" on page 5-16, and "PortfolioMAD Object Workflow" on page 6-15.

`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 be in a `NumSamples`-by-`NumAssets` matrix of `NumSamples` prices or returns at a given periodicity for a collection of `NumAssets` assets, a `table` or a `timetable`.

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: []
        MinNumAssets: []
        MaxNumAssets: []
           BoundType: []
```

```
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: []
       MinNumAssets: []
       MaxNumAssets: []
          BoundType: [4x1 categorical]
```

Simulate Multivariate Normal Asset Return Scenarios from Data for `PortfolioCVaR` Object Using Financial Timetable Data

To illustrate using the `simulateNormalScenariosByData` function with `AssetReturns` data continued in a `timetable` object, use the `CAPMuniverse.mat` which contains a `timetable` object (`AssetTimeTable`) for returns data.

```
load CAPMuniverse;
AssetsTimeTable.Properties
```

```
ans = 
  TimetableProperties with properties:

             Description: ''
                UserData: []
          DimensionNames: {'Time'  'Variables'}
           VariableNames: {'AAPL'  'AMZN'  'CSCO'  'DELL'  'EBAY'  'GOOG'  'HPQ'  'IBM'  'INTC'
    VariableDescriptions: {}
           VariableUnits: {}
      VariableContinuity: []
                RowTimes: [1471x1 datetime]
               StartTime: 03-Jan-2000
              SampleRate: NaN
                TimeStep: NaN
                  Events: []
        CustomProperties: No custom properties are set.
      Use addprop and rmprop to modify CustomProperties.
```

```
head(AssetsTimeTable,5)
```

```
       Time          AAPL        AMZN        CSCO        DELL        EBAY       GOOG      H
    ___________    _________   _________   _________   _________   _________    ____    _____

    03-Jan-2000     0.088805      0.1742    0.008775   -0.002353     0.12829    NaN       0.0
    04-Jan-2000    -0.084331    -0.08324    -0.05608    -0.08353   -0.093805    NaN     -0.07
    05-Jan-2000     0.014634    -0.14877   -0.003039    0.070984    0.066875    NaN     -0.00
    06-Jan-2000    -0.086538   -0.060072   -0.016619   -0.038847   -0.012302    NaN     -0.06
    07-Jan-2000     0.047368    0.061013      0.0587   -0.037708   -0.000964    NaN      0.02
```

Notice that GOOG has missing data (NaN) because it was not listed before Aug 2004. The simulateNormalScenariosByData function has a name-value pair argument '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, the estimateAssetMoments function uses the ECM algorithm to estimate asset moments. The simulateNormalScenariosByData function also accepts a name-value pair argument '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').

```
NumScenarios = 100;
r = PortfolioCVaR;
r = simulateNormalScenariosByData(r,AssetsTimeTable,NumScenarios,'DataFormat','Returns','MissingD
```

In addition, simulateNormalScenariosByData extracts asset names or identifiers from a timetable object when the name-value argument 'GetAssetList' is set to true (its default value is false). If the 'GetAssetList' value is true, the timetable column identifiers are used to set the AssetList property of the PortfolioCVaR object. To show this, the formation of the PortfolioCVaR object r is repeated with the 'GetAssetList' flag set to true.

```
r = simulateNormalScenariosByData(r,AssetsTimeTable,NumScenarios,'GetAssetList',true);
disp(r.AssetList)
```

```
    {'AAPL'}    {'AMZN'}    {'CSCO'}    {'DELL'}    {'EBAY'}    {'GOOG'}    {'HPQ'}    {'IBM'}
```

Estimate Mean and Covariance of Asset Returns 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. The market data, AssetsTimeTable, is a timetable of asset returns.

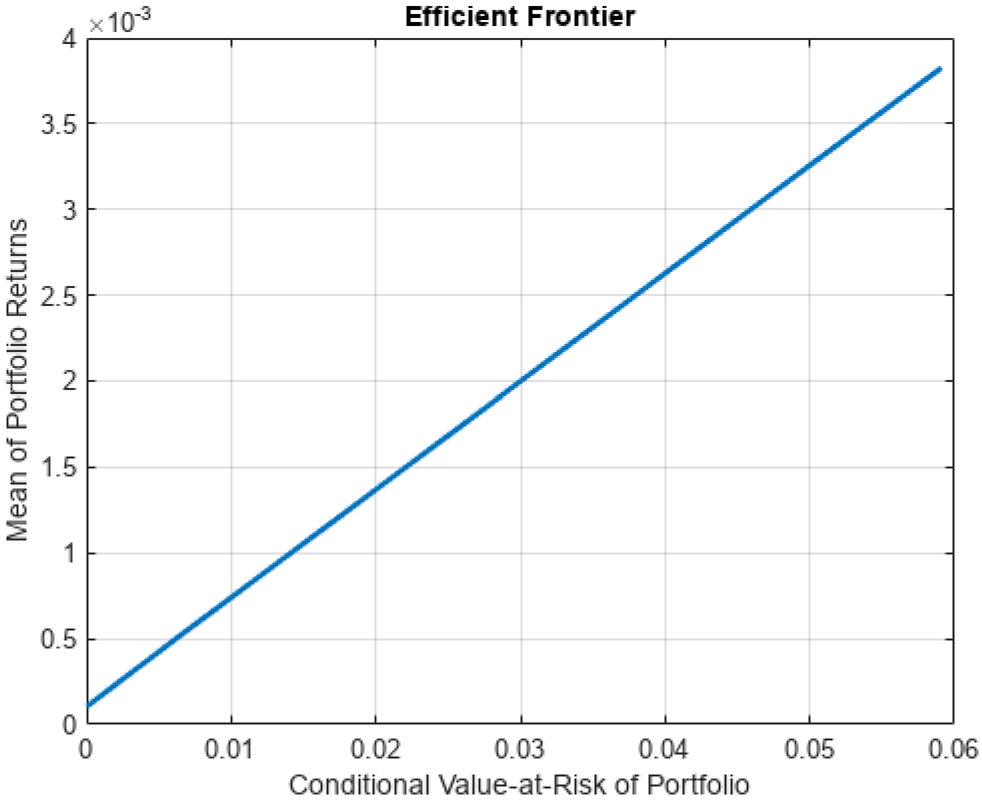
```
load CAPMuniverse

p = PortfolioCVaR('AssetList',Assets);
disp(p);
```

```
  PortfolioCVaR with properties:

             BuyCost: []
            SellCost: []
        RiskFreeRate: []
    ProbabilityLevel: []
            Turnover: []
         BuyTurnover: []
        SellTurnover: []
        NumScenarios: []
                Name: []
           NumAssets: 14
           AssetList: {'AAPL'  'AMZN'  'CSCO'  'DELL'  'EBAY'  'GOOG'  'HPQ'  'IBM'  'INTC'  'MSI
            InitPort: []
         AInequality: []
         bInequality: []
           AEquality: []
```

```
          bEquality: []
         LowerBound: []
         UpperBound: []
        LowerBudget: []
        UpperBudget: []
        GroupMatrix: []
         LowerGroup: []
         UpperGroup: []
             GroupA: []
             GroupB: []
         LowerRatio: []
         UpperRatio: []
       MinNumAssets: []
       MaxNumAssets: []
          BoundType: []
```

Simulate the scenarios from the timetable data for each of the assets from `CAPMuniverse.mat` and plot the efficient frontier.

```
p = simulateNormalScenariosByData(p,AssetsTimeTable,10000,'missingdata',true);
p = setDefaultConstraints(p);
p = setProbabilityLevel(p, 0.9);
plotFrontier(p);
```



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: []
    MinNumAssets: []
    MaxNumAssets: []
       BoundType: [4x1 categorical]
```

Estimate Mean and Covariance of Asset Returns from Market Data for a PortfolioMAD Object

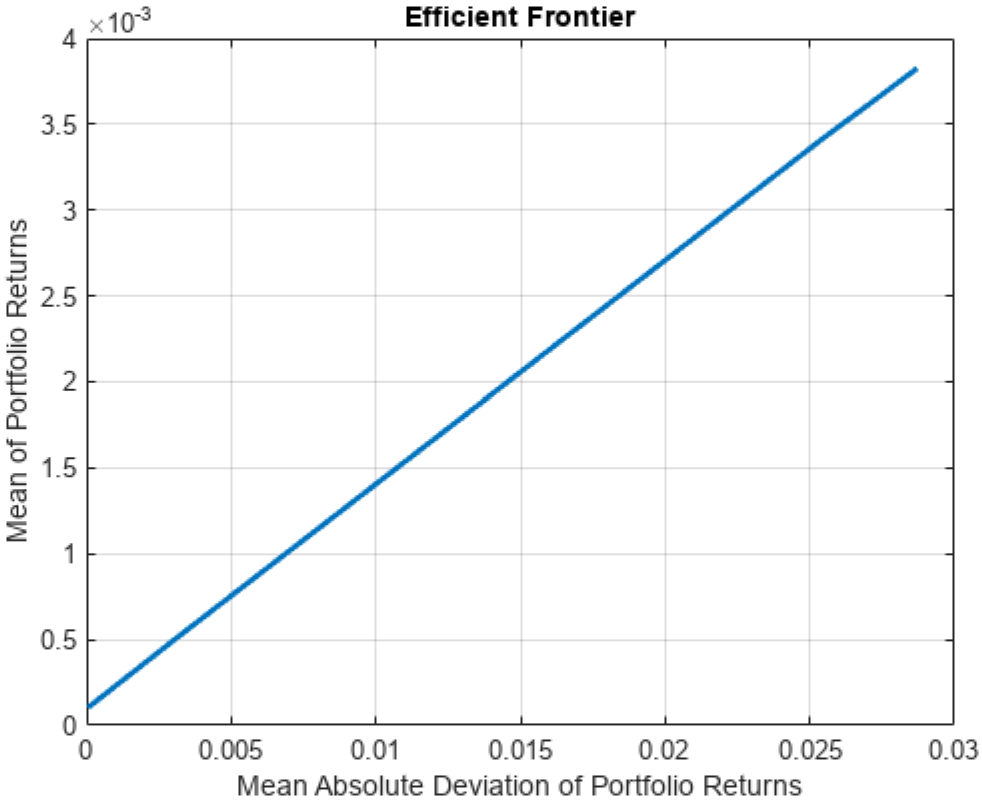
Create a PortfolioMAD object p and use the simulateNormalScenariosByData function with market data loaded from CAPMuniverse.mat to simulate multivariate normal asset return scenarios. The market data, AssetsTimeTable, is a timetable of asset returns.

```
load CAPMuniverse

p = PortfolioMAD('AssetList',Assets);
disp(p.AssetList');
```

```
    {'AAPL'  }
    {'AMZN'  }
    {'CSCO'  }
    {'DELL'  }
    {'EBAY'  }
    {'GOOG'  }
    {'HPQ'   }
    {'IBM'   }
    {'INTC'  }
    {'MSFT'  }
    {'ORCL'  }
    {'YHOO'  }
    {'MARKET'}
    {'CASH'  }
```

Simulate the scenarios from the timetable data for each of the assets from CAPMuniverse.mat and plot the efficient frontier.

```
p = simulateNormalScenariosByData(p,AssetsTimeTable,10000,'missingdata',true);
p = setDefaultConstraints(p);
plotFrontier(p);
```



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

Data Types: object

AssetReturns — Asset data that can be converted into asset returns

matrix | table | timetable

Asset data that can be converted into asset returns ([NumSamples-by-NumAssets] matrix), specified as a matrix, table, or timetable.

AssetReturns data can be:

- NumSamples-by-NumAssets matrix.
- Table of NumSamples prices or returns at a given periodicity for a collection of NumAssets assets

- Timetable object with `NumSamples` observations and `NumAssets` time series

Data Types: `double` | `table` | `timetable`

`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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `p = simulateNormalScenariosByData(p,RawData,NumScenarios,'DataFormat','Returns','MissingData',true,'GetAssetList',true)`

`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 as the comma-separated pair consisting of `'DataFormat'` and 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, specified as the comma-separated pair consisting of `'MissingData'` and 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 the comma-separated pair consisting of `'GetAssetList'` and a logical with a value of `true` or `false`.

- `false` — Do not extract or create asset names.
- `true` — Extract or create asset names from the table or timetable.

If a `table` or `timetable` is passed into this function using the `AssetReturns` argument and the `GetAssetList` flag is `true`, the column names from the `table` or `timetable` 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,Name,Value);
```

Version History

Introduced in R2012b

R2023a: `fints` support for `AssetReturns` argument removed

Behavior changed in R2023a

The `AssetReturns` argument no longer supports a `fints` object as a datatype.

See Also

`simulateNormalScenariosByMoments` | `rng`

Topics

“Asset Returns and Scenarios Using PortfolioCVaR Object” on page 5-36
“Asset Returns and Scenarios Using PortfolioMAD Object” on page 6-34

External Websites

Getting Started with Portfolio Optimization (4 min 13 sec)
CVaR Portfolio Optimization (4 min 56 sec)

simulateNormalScenariosByMoments

Simulate multivariate normal asset return scenarios from mean and covariance of asset returns

Syntax

```
obj = simulateNormalScenariosByMoments(obj,AssetMean,AssetCovar,NumScenarios)
obj = simulateNormalScenariosByMoments(obj,AssetMean,AssetCovarNumScenarios,
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. For details on the workflows, see "PortfolioCVaR Object Workflow" on page 5-16, and "PortfolioMAD Object Workflow" on page 6-15.

`obj = simulateNormalScenariosByMoments(obj,AssetMean,AssetCovarNumScenarios, 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: []
        MinNumAssets: []
        MaxNumAssets: []
           BoundType: [4x1 categorical]
```

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: []
    MinNumAssets: []
    MaxNumAssets: []
       BoundType: [4x1 categorical]
```

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

Data Types: object

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.
- If AssetCovar is not a symmetric positive semidefinite matrix, use nearcorr to create a positive semidefinite matrix for a correlation 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

`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);
```

Version History

Introduced in R2012b

See Also

`simulateNormalScenariosByData` | `rng` | `nearcorr` | `covarianceShrinkage` | `covarianceDenoising`

Topics

"Asset Returns and Scenarios Using PortfolioCVaR Object" on page 5-36
"Asset Returns and Scenarios Using PortfolioMAD Object" on page 6-34

External Websites

Getting Started with Portfolio Optimization (4 min 13 sec)
CVaR Portfolio Optimization (4 min 56 sec)

sharpe

Compute Sharpe ratio for one or more assets

Syntax

```
sharpe(Asset)
sharpe(Asset,Cash)
Ratio = sharpe(Asset,Cash)
```

Description

`sharpe(Asset)` computes Sharpe ratio for each asset.

`sharpe(Asset,Cash)` computes Sharpe ratio for each asset including the optional argument `Cash`.

`Ratio = sharpe(Asset,Cash)` computes Sharpe ratio for each asset including the optional argument `Cash`.

Examples

Compute Sharpe Ratio

This example shows how to compute the Sharpe ratio using the mean return of a cash asset as the return for the riskless asset.

Given asset return data and the riskless asset return, the Sharpe ratio is calculated:

```
load FundMarketCash
Returns = tick2ret(TestData);
Riskless = mean(Returns(:,3))
```

```
Riskless = 0.0017
```

```
Sharpe = sharpe(Returns, Riskless)
```

```
Sharpe = 1×3

    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 is calculated with the mean of cash returns. The Sharpe ratio can also be calculated with the cash return series as input for the riskless asset.

```
Sharpe = sharpe(Returns, Returns(:,3))
```

```
Sharpe = 1×3
```

```
    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 4-107.

Input Arguments

`Asset` — Asset returns

matrix

Asset returns, specified as a `NUMSAMPLES x NUMSERIES` matrix with `NUMSAMPLES` observations of asset returns for `NUMSERIES` asset return series.

Data Types: `double`

`Cash` — Riskless asset

`0` (default) | numeric | vector

(Optional) Riskless asset, specified as a 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`.

Data Types: `double`

Output Arguments

`Ratio` — Sharpe ratios

vector

Sharpe ratios, returned as 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.

Version History

Introduced in R2006b

References

[1] Sharpe, W. F. "Mutual Fund Performance." *Journal of Business.* Vol. 39, No. 1, Part 2, January 1966, pp. 119–138.

See Also

`inforatio` | `portalpha`

Topics

"Performance Metrics Illustration" on page 7-3
"Performance Metrics Overview" on page 7-2

stochosc

Stochastic oscillator

Syntax

```
percentKnD = stochosc(Data)
percentKnD = stochosc( ___ ,Name,Value)
```

Description

`percentKnD = stochosc(Data)` calculates the stochastic oscillator.

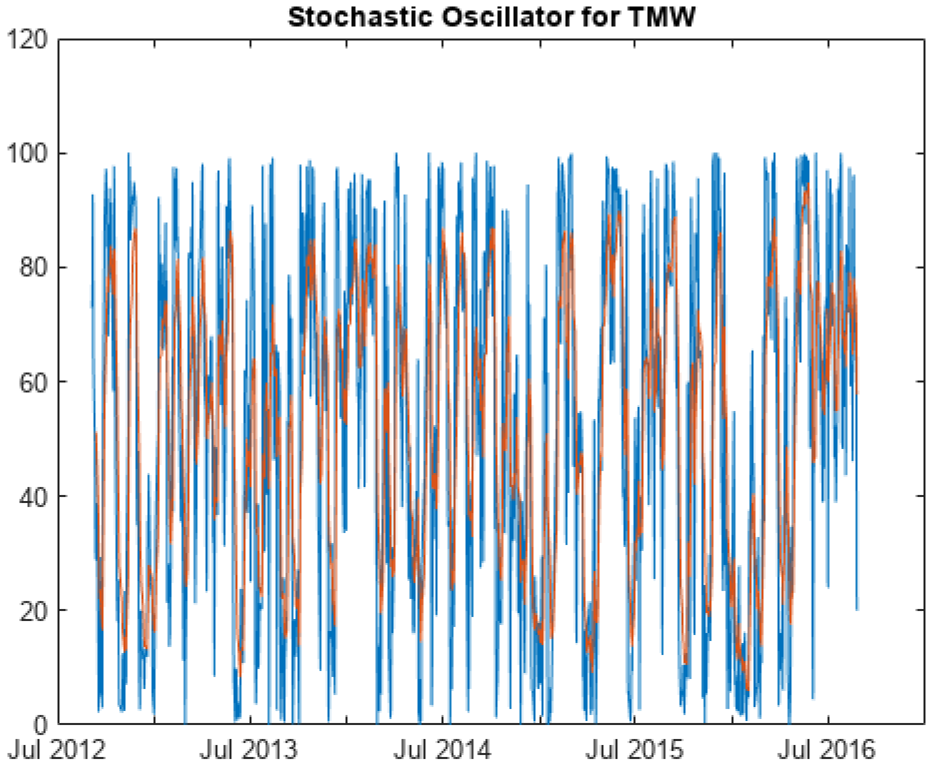
`percentKnD = stochosc( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Calculate the Stochastic Oscillator for a Stock

Load the file `SimulatedStock.mat`, which provides a timetable (`TMW`) for financial data for TMW stock.

```
load SimulatedStock.mat
oscillator = stochosc(TMW,'NumPeriodsD',7,'NumPeriodsK',10,'Type','exponential');
plot(oscillator.Time,oscillator.FastPercentK,oscillator.Time,oscillator.FastPercentD)
title('Stochastic Oscillator for TMW')
```



Input Arguments

`Data` — Data with high, low, open, close information

matrix | table | timetable

Data with high, low, open, close information, specified as a matrix, table, or timetable. For matrix input, `Data` is an `M`-by-`3` matrix of high, low, and closing prices stored in the corresponding columns, respectively. Timetables and tables with `M` rows must contain variables named `'High'`, `'Low'`, and `'Close'` (case insensitive).

Data Types: `double` | `table` | `timetable`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `percentKnD = stochosc(TMW,'NumPeriodsD',10,'NumPeriodsK',3,'Type','exponential')`

`NumPeriodsK` — Period difference for PercentK

`10` (default) | positive integer

Period difference for PercentK, specified as the comma-separated pair consisting of `'NumPeriodsK'` and a scalar positive integer.

Data Types: `double`

`NumPeriodsD` — Length of moving average in periods for PercentD
`3` (default) | positive integer

Length of moving average in periods for PercentD, specified as the comma-separated pair consisting of `'NumPeriodsD'` and a scalar positive integer.

Data Types: `double`

Type — Moving average method for PercentD calculation
`'e'` (exponential) (default) | character vector with values `'exponential'` or `'triangular'`

Moving average method for PercentD calculation, specified as the comma-separated pair consisting of `'Type'` and a character vector with a value of:

- `'exponential'` – Exponential moving average is a weighted 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%.
- `'triangular'` – Triangular moving average is a double-smoothing of the data. The first simple moving average is calculated and then a second simple moving average is calculated on the first moving average with the same window size.

Data Types: `char`

Output Arguments

`percentKnD` — PercentK and PercentD
matrix | table | timetable

PercentK and PercentD, returned with the same number of rows (`M`) and type (matrix, table, or timetable) as the input `Data`.

More About

Stochastic Oscillator

The stochastic oscillator calculates the Fast PercentK (F%K), Fast PercentD (F%D), Slow PercentK (S %K), and Slow PercentD (S%D) from the series of high, low, and closing stock prices.

By default, the stochastic oscillator is based on 10-period difference for PercentK and a 3-period exponential moving average for PercentD.

Version History

Introduced before R2006a

R2023a: `fints` support removed for `Data` input argument
Behavior changed in R2023a

`fints` object support for the `Data` input argument is removed.

R2022b: Support for negative price data
Behavior changed in R2022b

The `Data` input accepts negative prices.

References

[1] Achelis, S. B. *Technical Analysis from A to Z.* Second Edition. McGraw-Hill, 1995, pp. 268–271.

See Also

`timetable` | `table` | `chaikosc`

Topics

"Use Timetables in Finance" on page 11-7
"Convert Financial Time Series Objects (fints) to Timetables" on page 11-2

targetreturn

Portfolio weight accuracy

Syntax

```
return = targetreturn(Universe,Window,Offset,Weights)
```

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`.

Input Arguments

`Universe` — Total return data for a group of securities

array

Total return data for a group of securities, specified as a number of observations (`NUMOBS`) by number of assets plus one (`NASSETS + 1`) array. Each row represents an observation. Column 1 contains MATLAB serial date numbers. The remaining columns contain the total return data for each security.

Data Types: `double`

`Window` — Number of data periods used to calculate frontier

integer

Number of data periods used to calculate frontier, specified as an integer.

Data Types: `double`

`Offset` — Number of periods to increment when each frontier is generated

integer

Number of periods to increment when each frontier is generated , specified as an integer.

Data Types: `double`

`Weights` — Asset allocation weights needed to obtain the target rate of return

matrix

Asset allocation weights needed to obtain the target rate of return, specified as a number of assets (NASSETS) by number of curves (NCURVES) matrix.

Data Types: `double`

Output Arguments

`return` — Target return

numeric

Target return, returned as a numeric value for each `Window` of data and given portfolio `Weights`.

Version History

Introduced before R2006a

See Also

`selectreturn` | `frontier` | `portopt`

Topics

"Portfolio Construction Examples" on page 3-5
"Portfolio Optimization Functions" on page 3-3

taxedrr

After-tax rate of return

Syntax

`Return = taxedrr(PreTaxReturn,TaxRate)`

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
```

Input Arguments

`PreTaxReturn` — Normal rate of return
decimal

Normal rate of return, specified as a decimal.

Data Types: `double`

`TaxRate` — Tax rate
decimal

Tax rate, specified as a decimal.

Data Types: `double`

Output Arguments

`Return` — After-tax rate of return
decimal

After-tax rate of return, returned as a decimal.

Version History

Introduced before R2006a

See Also

`effrr` | `irr` | `mirr` | `nomrr` | `xirr`

Topics

"Analyzing and Computing Cash Flows" on page 2-11

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(2002,10,1);
Maturity = datetime(2003,3,31);
[BEYield MMYield] = tbilldisc2yield(Discount, Settle, Maturity)
```

```
BEYield = 0.0517

MMYield = 0.0510
```

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

datetime array | string array | date character vector

Settlement date of the Treasury bill, specified as a scalar or a `NTBILLS`-by-`1` vector using a datetime array, string array, or date character vectors. `Settle` must be earlier than `Maturity`.

To support existing code, `tbilldisc2yield` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Maturity` — Maturity date of Treasury bill

datetime array | string array | date character vector

Maturity date of the Treasury bill, specified as a scalar or a `NTBILLS`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `tbilldisc2yield` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `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.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although tbilldisc2yield supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

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

tbillyield2disc | zeroyield | datetime

Topics

"Computing Treasury Bill Price and Yield" on page 2-26
"Bond Portfolio Optimization Using Portfolio Object" on page 10-30
"Treasury Bills Defined" on page 2-25

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 = 2×1

   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(2002,1,2);datetime(2002,3,1)];
Maturity = [datetime(2002,6,30);datetime(2002,6,30)];
Price = tbillprice(Rate, Settle, Maturity, Type)
```

```
Price = 2×1

   97.8408
   98.4539
```

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

datetime array | string array | date character vector

Settlement date of the Treasury bill, specified as a scalar or a `NTBILLS`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `tbillprice` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Maturity` — Maturity date of Treasury bill

datetime array | string array | date character vector

Maturity date of the Treasury bill, specified as a scalar or a `NTBILLS`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `tbillprice` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `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.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although tbillprice supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)

y =

        2021
```

There are no plans to remove support for serial date number inputs.

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

tbillyield | zeroprice | datetime

Topics

"Computing Treasury Bill Price and Yield" on page 2-26
"Bond Portfolio Optimization Using Portfolio Object" on page 10-30
"Treasury Bills Defined" on page 2-25

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.

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 = 2×1

    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(2002,1,3);
SaleDate = datetime(2002,2,3);
Maturity = datetime(2002,4,3);
TBEDiscount = tbillrepo(RepoRate, InitialDiscount,...
PurchaseDate, SaleDate, Maturity)
```

```
TBEDiscount = 2×1

    0.0491
    0.0478
```

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
datetime array | string array | date character vector

Date the Treasury bill is purchased, specified as a scalar or a `NTBILLS`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `tbillrepo` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`SaleDate` — Date Treasury bill repurchase term is due
datetime array | string array | date character vector

Date the Treasury bill repurchase term is due, specified as a scalar or a `NTBILLS`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `tbillrepo` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Maturity` — Maturity date of Treasury bill
datetime array | string array | date character vector

Maturity date of the Treasury bill, specified as a scalar or a `NTBILLS`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `tbillrepo` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `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.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended
Not recommended starting in R2022b

Although `tbillrepo` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

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

`tbillyield` | `tbillprice` | `tbillval01` | `datetime`

Topics
"Computing Treasury Bill Price and Yield" on page 2-26
"Bond Portfolio Optimization Using Portfolio Object" on page 10-30
"Treasury Bills Defined" on page 2-25

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(2003,3,1);
Maturity = datetime(2003,6,30);
[Val01Disc, Val01MMY, Val01BEY] = tbillval01(Settle, Maturity)
```

```
Val01Disc = 0.0034

Val01MMY = 0.0034

Val01BEY = 0.0033
```

Input Arguments

Settle — Settlement date of Treasury bill

datetime array | string array | date character vector

Settlement date of the Treasury bill, specified as a scalar or a NTBILLS-by-1 vector using a datetime array, string array, or date character vectors. Settle must be earlier than Maturity.

To support existing code, tbillval01 also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

Maturity — Maturity date of Treasury bill

datetime array | string array | date character vector

Maturity date of the Treasury bill, specified as a scalar or a NTBILLS-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, tbillval01 also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | 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.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although tbillval01 supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)

y =

        2021
```

There are no plans to remove support for serial date number inputs.

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

`tbillyield2disc` | `zeroyield` | `datetime`

Topics

"Computing Treasury Bill Price and Yield" on page 2-26
"Bond Portfolio Optimization Using Portfolio Object" on page 10-30
"Treasury Bills Defined" on page 2-25

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
```

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

datetime array | string array | date character vector

Settlement date of the Treasury bill, specified as a scalar or a `NTBILLS`-by-`1` vector using a datetime array, string array, or date character vectors. `Settle` must be earlier than `Maturity`.

To support existing code, `tbillyield` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Maturity` — Maturity date of Treasury bill

datetime array | string array | date character vector

Maturity date of the Treasury bill, specified as a scalar or a `NTBILLS`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `tbillyield` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `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.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although tbillyield supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

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

zeroyield | tbillprice | tbilldisc2yield | tbillyield2disc | datetime

Topics

“Computing Treasury Bill Price and Yield” on page 2-26
“Bond Portfolio Optimization Using Portfolio Object” on page 10-30
“Treasury Bills Defined” on page 2-25

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(2002,10,1);
Maturity = datetime(2003,3,31);

Discount = tbillyield2disc(Yield,Settle,Maturity)
```

```
Discount = 0.0485
```

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

datetime array | string array | date character vector

Settlement date of the Treasury bill, specified as a scalar or a `NTBILLS`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `tbillyield2disc` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Maturity` — Maturity date of Treasury bill

datetime array | string array | date character vector

Maturity date of the Treasury bill, specified as a scalar or a `NTBILLS`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `tbillyield2disc` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `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.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although `tbillyield2disc` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)

y =

        2021
```

There are no plans to remove support for serial date number inputs.

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

`tbilldisc2yield` | `datetime`

Topics

“Computing Treasury Bill Price and Yield” on page 2-26
“Bond Portfolio Optimization Using Portfolio Object” on page 10-30
“Treasury Bills Defined” on page 2-25

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 = 3×5
10^5 ×

         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=3×5 table
    CouponRate     Maturity       Bid      Asked     AskYield
    __________    ___________    ______    ______    ________

        0         02-Jan-1998    99.854    99.855      0.053
        0         05-Feb-1998    99.344    99.349     0.0544
        0         05-Mar-1998    98.942    98.946      0.054


Settle = 3x1 datetime
   22-Dec-1997
   22-Dec-1997
   22-Dec-1997
```

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 datetime, string, date character vector, or 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 a datetime array, string array, date character vectors, or serial date numbers.
- 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 | char | string | datetime | 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. The format of the dates matches the format used for `Maturity` in `TBillMatrix` (datetime array, string array, date character vector, or serial date number).
- `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. Use the function `datestr` to convert serial date numbers to formatted date character vectors. `Settle` is returned as a datetime array only if the input `TBillMatrix` is a table containing datetime arrays for `Maturity` in the first column.

Version History

Introduced before R2006a

See Also

`tr2bonds` | `datetime`

Topics

"Term Structure of Interest Rates" on page 2-29
"Computing Treasury Bill Price and Yield" on page 2-26
"Bond Portfolio Optimization Using Portfolio Object" on page 10-30
"Treasury Bills Defined" on page 2-25

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
   16-Oct-2002
   15-Oct-2003
   20-Oct-2004


EndDates = 3x1 datetime
   16-Jan-2003
   15-Jan-2004
   20-Jan-2005
```

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'`.

Version History

Introduced before R2006a

See Also

`tr2bonds` | `datetime`

Topics
"Handle and Convert Dates" on page 2-2
"Trading Calendars User Interface" on page 12-2
"UICalendar User Interface" on page 12-4

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 = 2×1

  101.7812
  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.

Version History

Introduced before R2006a

See Also

`dec2thirtytwo`

tick2ret

Convert price series to return series

Syntax

```
[Returns,Intervals] = tick2ret(Data)
[Returns,Intervals] = tick2ret( ___ ,Name,Value)
```

Description

`[Returns,Intervals] = tick2ret(Data)` computes asset returns for `NUMOBS` price observations of `NASSETS` assets.

`[Returns,Intervals] = tick2ret( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Convert Price Series to Return Series

Load the file `SimulatedStock.mat`, which provides a timetable (`TMW`) for financial data for TMW stock. Then convert a price series to a return series, given the first 10 periodic returns of `TMW`.

```
load SimulatedStock.mat

TMW_Close = TMW(1:10,'Close');
[Returns,Intervals] = tick2ret(TMW_Close)
```

```
Returns=9×1 timetable
       Time           Close
    ___________    ___________

    05-Sep-2012      0.0017955
    06-Sep-2012       0.013741
    07-Sep-2012      -0.022591
    10-Sep-2012      -0.011557
    11-Sep-2012      -0.014843
    12-Sep-2012     -0.0012384
    13-Sep-2012      0.0081628
    14-Sep-2012    -0.00051245
    17-Sep-2012       -0.02902


Intervals = 9x1 duration
   24:00:00
   24:00:00
   24:00:00
   72:00:00
   24:00:00
   24:00:00
   24:00:00
   24:00:00
   72:00:00
```

Convert Price Series to Return Series Using `datetime` Input

Use `datetime` input to convert a price series to a 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 = datetime({'1/1/2015','1/7/2015','1/16/2015','1/28/2015'},'InputFormat','MM/dd/uuuu')
[Returns,Intervals] = tick2ret(TickSeries,'TickTimes',TickTimes)
```

```
Returns = 3×2

    0.1000    0.1250
    0.0455   -0.0222
   -0.0435    0.0341


Intervals = 3x1 duration
   144:00:00
   216:00:00
   288:00:00
```

Input Arguments

`Data` — Data for asset prices

matrix | table | timetable

Data for asset prices, specified as a `NUMOBSNASSETS` matrix, `table`, or `timetable`. Prices across a given row are assumed to occur at the same time for all columns, and each column is a price series of an individual asset.

Data Types: `double` | `table` | `timetable`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Returns,Intervals] = tick2ret(TickSeries,'TickTimes',TickTimes)`

`TickTimes` — Observation times associated with prices

sequential observation times from 1,2,...NUMOBS assumed for all assets (default) | vector

Observation times associated with prices, specified as the comma-separated pair consisting of 'TickTimes' and a NUMOBS element column vector of monotonically increasing observation times associated with the prices in Data. Times are taken either as serial date numbers (day units), date strings, datetime arrays, or as decimal numbers in arbitrary units (for example, yearly).

Note If the input Data type is a timetable, the row times information in the timetable overwrites the TickTimes input.

Data Types: double | datetime | string

Method — Method to convert asset prices to returns

'Simple' (default) | character vector with value of 'Simple' or 'Continuous' | string with value of "Simple" or "Continuous"

Method to convert asset prices to returns, specified as the comma-separated pair consisting of 'Method' and a string or character vector indicating the method to convert asset prices to returns.

If the method is 'Simple', then simple periodic returns at time *t* are computed as:

```
Returns(t) = Data(t)/Data(t-1) - 1.
```

If the method is 'Continuous', the continuous returns are computed as:

```
Returns(t) = log(Data(t)/Data(t-1)).
```

Data Types: char | string

Output Arguments

Returns — Time series array of asset returns

matrix | table | timetable

Time series array of asset returns, returned as a NUMOBS-1-by-NASSETS array of asset returns with the same type (matrix, table, or timetable) as the input Data. The first row contains the oldest returns and the last row contains the most recent. Returns across a given row are assumed to occur at the same time for all columns, and each column is a return series of an individual asset.

Intervals — Interval times between successive prices

vector

Interval times between successive prices, returned as a NUMOBS-1 length column vector where Intervals(*t*) = TickTimes(*t*) - TickTimes(*t* - 1).

Version History

Introduced before R2006a

R2022b: Support for negative price data

Behavior changed in R2022b

The Data input accepts negative prices.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports input `Data` that is specified as a tall column vector, a tall table, or a tall timetable. For more information, see `tall` and “Tall Arrays”.

See Also

`ret2tick` | `timetable` | `table` | `datetime`

Topics

“Use Timetables in Finance” on page 11-7
“Returns with Negative Prices” on page 2-32

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 = datetime(2002,9,1);
Dates = [datetime(2005,8,31) ; datetime(2006,2,28) ; datetime(2006,6,15) ; datetime(2006,12,31)]
Compounding = 2;
Basis = 0;
EndMonthRule = 1;
TFactors = date2time(Settle, Dates, Compounding, Basis,...
EndMonthRule)
```

```
TFactors = 4×1

    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 = 4x1 datetime
   31-Aug-2005
   28-Feb-2006
   15-Jun-2006
```

```
31-Dec-2006
```

Input Arguments

`Settle` — Settlement date

datetime scalar | string scalar | date character vector

Settlement date, specified as a scalar datetime, string, or date character vector.

To support existing code, `time2date` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `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 2-16.

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 a datetime array.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although `time2date` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`cftimes` | `date2time` | `datetime` | `cfamounts`

Topics

"Handle and Convert Dates" on page 2-2

tmfactor

Time factors of arbitrary dates

Syntax

```
TMFactors = tmfactor(Settle,Maturity)
```

Description

`TMFactors = tmfactor(Settle,Maturity)` determines the time factors from a vector of Settlement dates to a vector of Maturity dates.

Examples

Find Time Factors of `Settle` and `Maturity` Dates

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 datetimes.

```
TFactors = tmfactor(datetime(2015,1,1),datetime(2016,1,1))
```

```
TFactors = 2
```

Input Arguments

`Settle` — Settlement date

datetime array | string array | date character vector

Settlement date, specified as a scalar or a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

Note `Settle` must be earlier than `Maturity`.

To support existing code, `tmfactor` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Maturity` — Maturity date

datetime array | string array | date character vector

Maturity date, specified as a scalar or a `NINST`-by-`1` vector using a datetime array, string array, or date character vectors.

Data Types: char | string | datetime

Output Arguments

TMFactors — Time factors from a vector of Settle dates to a vector of Maturity dates

scalar numeric

Time factors from a vector of Settle dates to a vector of Maturity dates, returned as a scalar numeric.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although tmfactor supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

cfamounts | cftimes | datetime

Topics

"Analyzing and Computing Cash Flows" on page 2-11

todecimal

Fractional to decimal conversion

Syntax

```
usddec = todecimal(quote)
usddec = todecimal( ___ ,fracpart)
```

Description

`usddec = todecimal(quote)` returns the decimal equivalent, `usddec`, of a security whose price is normally quoted as a whole number and a fraction (`quote`).

`usddec = todecimal( ___ ,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

Convert Stock Quote from Fractinal to Decimal

Often secuirty 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 = 100.1562
```

You can change the defualt denominator of 32 by using the `fracpart` optional argument with a value of `16`.

```
usddec = todecimal(97.04, 16)
```

```
usddec = 97.2500
```

Input Arguments

`quote` — Quoted security price
scalar numeric

Quoted security price, specified as a scalar numeric or an `NPORTS`-by-`1` vector.

Note The convention of using `.` (period) as a substitute for `:` (colon) in the input is adopted from Excel software.

Data Types: `double`

`fracpart` — Fractional base (denominator) with which the security is normally quoted

32 (default) | scalar numeric

(Optional) Fractional base (denominator) with which the security is normally quoted, specified as a scalar numeric.

Data Types: `double`

Output Arguments

`usddec` — Fractional to decimal conversion

numeric

Fractional to decimal conversion, returned as a numeric decimal.

Version History

Introduced before R2006a

See Also

`toquoted`

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

Convert Decimal to Fraction

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 = 101.0500
```

The answer is interpreted as 101 5/8.

The convention of using . (period) as a substitute for : (colon) in the output is adopted from Excel ® software.

Input Arguments

`usddec` — Equity price
numeric

Equity price, specified as a numeric decimal.

Data Types: `double`

`fracpart` — Fractional bases used for quoting equity prices in the United States
32 (default) | numeric with denominator value of 2, 4, 8, 16, or 32

Fractional bases used for quoting equity prices in the United States, specified as a scalar numeric.

Data Types: `double`

Output Arguments

`quote` — Fractional equivalent
numeric

Fractional equivalent, returned as a numeric. For example, bond prices are quoted in fractional form based on a 32nd. If you see the quoted price is `100:05`, it means 100 5/32 which is equivalent to 100.15625.

Version History

Introduced before R2006a

See Also

`todecimal`

convert2daily

Aggregate timetable data to daily periodicity

Syntax

```
TT2 = convert2daily(TT1)
TT2 = convert2daily(TT1,Name,Value)
```

Description

`TT2 = convert2daily(TT1)` aggregates data (for example, high-frequency and intra-day data) to a daily periodicity.

`TT2 = convert2daily(TT1,Name,Value)` uses additional options specified by one or more name-value arguments.

Examples

Aggregate Prices and Logarithmic Returns to Daily Periodicity

Apply separate aggregation methods to related variables in a `timetable` while maintaining consistency between aggregated results for a daily periodicity.

Load a timetable (`TT`) of simulated stock price data and corresponding logarithmic returns. The data stored in `TT` is recorded at various times throughout the day on New York Stock Exchange (NYSE) business days from January 1, 2018, to December 31, 2020. The timetable `TT` also includes NYSE business calendar awareness. If your timetable does not account for nonbusiness days (weekends, holidays, and market closures), add business calendar awareness by using `addBusinessCalendar` first.

```
load('SimulatedStock.mat','TT');
head(TT)
```

```
            Time            Price     Log_Return
    ____________________    ______    __________

    02-Jan-2018 11:52:11    100.71     0.0070749
    02-Jan-2018 13:23:09    103.11      0.023551
    02-Jan-2018 14:45:30    100.24     -0.028229
    02-Jan-2018 15:30:48    101.37       0.01121
    03-Jan-2018 10:02:21    101.81     0.0043311
    03-Jan-2018 11:22:37    100.17      -0.01624
    03-Jan-2018 14:45:20     99.66    -0.0051043
    03-Jan-2018 14:55:39    100.12     0.0046051
```

Aggregate prices and logarithmic returns to a daily periodicity. To maintain consistency between prices and returns, for any given trading day, aggregate the prices by reporting the last recorded price by using `"lastvalue"` and aggregate the returns by summing all logarithmic returns by using `"sum"`.

```
tt = convert2daily(TT,'Aggregation',["lastvalue" "sum"]);
head(tt)

       Time        Price     Log_Return
    ___________    ______    __________

    02-Jan-2018    101.37     0.013607
    03-Jan-2018    100.12    -0.012408
    04-Jan-2018    106.76     0.064214
    05-Jan-2018    112.78     0.054856
    08-Jan-2018    119.07     0.054273
    09-Jan-2018    119.46      0.00327
    10-Jan-2018    124.44     0.040842
    11-Jan-2018    125.63    0.0095174
```

To verify consistency, examine the input and output timetables for January 2 and 3, 2018.

```
TT(1:8,:)  % Input data for 02-Jan-2018 and 03-Jan-2018

ans=8×2 timetable
            Time             Price     Log_Return
    ____________________    ______    __________

    02-Jan-2018 11:52:11    100.71     0.0070749
    02-Jan-2018 13:23:09    103.11      0.023551
    02-Jan-2018 14:45:30    100.24     -0.028229
    02-Jan-2018 15:30:48    101.37       0.01121
    03-Jan-2018 10:02:21    101.81     0.0043311
    03-Jan-2018 11:22:37    100.17      -0.01624
    03-Jan-2018 14:45:20     99.66    -0.0051043
    03-Jan-2018 14:55:39    100.12     0.0046051
```

```
tt(1:2,:)  % Return aggregated results

ans=2×2 timetable
       Time        Price     Log_Return
    ___________    ______    __________

    02-Jan-2018    101.37     0.013607
    03-Jan-2018    100.12    -0.012408
```

For each business day in `TT`, notice that the output aggregated price is the last price of the day and that the aggregated return is the sum of all logarithmic returns. Also, the aggregated returns are consistent with aggregated prices.

For example, the aggregated return for January 3, 2018, is `-0.012408`, which is the logarithmic return associated with the last prices recorded on January 2 and 3, 2018 (that is, `-0.012408` = `log(100.12)` - `log(101.37)`).

The dates of the aggregated results are whole dates that indicate the dates for which aggregated results are reported.

Input Arguments

TT1 — Data to aggregate to daily periodicity
timetable

Data to aggregate to a daily periodicity, specified as a timetable.

Each variable can be a numeric vector (univariate series) or numeric matrix (multivariate series).

Note

- NaNs indicate missing values.
- Timestamps must be in ascending or descending order.

By default, all days are business days. If your timetable does not account for nonbusiness days (weekends, holidays, and market closures), add business calendar awareness by using addBusinessCalendar first. For example, the following command adds business calendar logic to include only NYSE business days.

```
TT = addBusinessCalendar(TT);
```

Data Types: timetable

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: TT2 = convert2daily(TT1,'Aggregation',["lastvalue" "sum"])

Aggregation — Intra-day aggregation method for data in TT1
"lastvalue" (default) | "sum" | "prod" | "mean" | "min" | "max" | "firstvalue" | character vector | function handle | string vector | cell vector of character vectors or function handles

Intra-day aggregation method for TT1 defining how data is aggregated over business days, specified as one of the following methods, a string vector of methods, or a length numVariables cell vector of methods, where numVariables is the number of variables in TT1.

- "sum" — Sum the values in each year or day.
- "mean" — Calculate the mean of the values in each year or day.
- "prod" — Calculate the product of the values in each year or day.
- "min" — Calculate the minimum of the values in each year or day.
- "max" — Calculate the maximum of the values in each year or day.
- "firstvalue" — Use the first value in each year or day.
- "lastvalue" — Use the last value in each year or day.
- @customfcn — A custom aggregation method that accepts a timetable and returns a numeric scalar (for univariate series) or row vector (for multivariate series). The function must accept empty inputs [].

If you specify a single method, convert2daily applies the specified method to all time series in TT1. If you specify a string vector or cell vector aggregation, convert2daily applies aggregation(*j*) to TT1(:,*j*); convert2daily applies each aggregation method one at a time (for more details, see retime). For example, consider a daily timetable representing TT1 with three variables.

```
        Time            AAA       BBB           CCC
____________________    ______    ______    ________________
01-Jan-2018 09:45:47    100.00    200.00    300.00    400.00
01-Jan-2018 12:48:09    100.03    200.06    300.09    400.12
02-Jan-2018 10:27:32    100.07    200.14    300.21    400.28
02-Jan-2018 12:46:09    100.08    200.16    300.24    400.32
02-Jan-2018 14:14:13    100.25    200.50    300.75    401.00
02-Jan-2018 15:52:31    100.19    200.38    300.57    400.76
03-Jan-2018 09:47:11    100.54    201.08    301.62    402.16
03-Jan-2018 11:24:23    100.59    201.18    301.77    402.36
03-Jan-2018 14:41:17    101.40    202.80    304.20    405.60
03-Jan-2018 16:00:00    101.94    203.88    305.82    407.76
04-Jan-2018 09:55:51    102.53    205.06    307.59    410.12
04-Jan-2018 10:07:12    103.35    206.70    310.05    413.40
04-Jan-2018 14:26:23    103.40    206.80    310.20    413.60
05-Jan-2018 13:13:12    103.91    207.82    311.73    415.64
05-Jan-2018 14:57:53    103.89    207.78    311.67    415.56
```

The corresponding default daily results representing TT2 (where the 'lastvalue' is reported for each day) are as follows.

```
  Time         AAA       BBB           CCC
___________    ______    ______    ________________
01-Jan-2018    100.03    200.06    300.09    400.12
02-Jan-2018    100.19    200.38    300.57    400.76
03-Jan-2018    101.94    203.88    305.82    407.76
04-Jan-2018    103.40    206.80    310.20    413.60
05-Jan-2018    103.89    207.78    311.67    415.56
```

All methods omit missing data (NaNs) in direct aggregation calculations on each variable. However, for situations in which missing values appear in the first row of TT1, missing values can also appear in the aggregated results TT2. To address missing data, write and specify a custom aggregation method (function handle) that supports missing data.

Data Types: char | string | cell | function_handle

Output Arguments

TT2 — Daily data

timetable

Daily data, returned as a timetable. The time arrangement of TT1 and TT2 are the same.

If a variable of TT1 has no records for a business day within the sampling time span, convert2daily returns a NaN for that variable and business day in TT2.

The first date in TT2 is the first business date on or after the first date in TT1. The last date in TT2 is the last business date on or before the last date in TT1.

Version History

Introduced in R2021a

See Also

`convert2weekly` | `convert2monthly` | `convert2quarterly` | `convert2semiannual` | `convert2annual` | `timetable` | `addBusinessCalendar`

Topics

"Resample and Aggregate Data in Timetable"
"Combine Timetables and Synchronize Their Data"
"Retime and Synchronize Timetable Variables Using Different Methods"

convert2weekly

Aggregate timetable data to weekly periodicity

Syntax

```
TT2 = convert2weekly(TT1)
TT2 = convert2weekly(___,Name,Value)
```

Description

`TT2 = convert2weekly(TT1)` aggregates data (for example, data recorded daily) to a weekly periodicity.

`TT2 = convert2weekly(___,Name,Value)` uses additional options specified by one or more name-value arguments.

Examples

Aggregate Timetable Daily Data to Weekly Periodicity

Apply separate aggregation methods to related variables in a `timetable` while maintaining consistency between aggregated results when converting from a daily to a weekly periodicity. You can use `convert2weekly` to aggregate both intra-daily data and aggregated daily data. These methods result in equivalent weekly aggregates.

Load a timetable (`TT`) of simulated stock price data and corresponding logarithmic returns. The data stored in `TT` is recorded at various times throughout the day on New York Stock Exchange (NYSE) business days from January 1, 2018, to December 31,2020. The timetable `TT` also includes NYSE business calendar awareness. If your timetable does not account for nonbusiness days (weekends, holidays, and market closures), add business calendar awareness by using `addBusinessCalendar` first.

```
load('SimulatedStock.mat','TT');
head(TT)
```

```
            Time            Price     Log_Return
    ____________________    ______    __________

    02-Jan-2018 11:52:11    100.71     0.0070749
    02-Jan-2018 13:23:09    103.11      0.023551
    02-Jan-2018 14:45:30    100.24     -0.028229
    02-Jan-2018 15:30:48    101.37       0.01121
    03-Jan-2018 10:02:21    101.81     0.0043311
    03-Jan-2018 11:22:37    100.17      -0.01624
    03-Jan-2018 14:45:20     99.66    -0.0051043
    03-Jan-2018 14:55:39    100.12     0.0046051
```

Use `convert2daily` to aggregate intra-daily prices and returns to daily periodicity. To maintain consistency between prices and returns, for any given trading day, aggregate prices by reporting the last recorded price with "lastvalue" and aggregate returns by summing all logarithmic returns with "sum".

```
TT1 = convert2daily(TT,'Aggregation',["lastvalue" "sum"]);
head(TT1)
```

```
       Time         Price     Log_Return
    ___________    ______    __________

    02-Jan-2018    101.37     0.013607
    03-Jan-2018    100.12    -0.012408
    04-Jan-2018    106.76     0.064214
    05-Jan-2018    112.78     0.054856
    08-Jan-2018    119.07     0.054273
    09-Jan-2018    119.46      0.00327
    10-Jan-2018    124.44     0.040842
    11-Jan-2018    125.63    0.0095174
```

Use convert2weekly to aggregate the data to a weekly periodicity and compare the results of two different aggregation approaches. The first approach computes weekly results by aggregating the daily aggregates and the second approach computes weekly results by directly aggregating the original intra-daily data.

```
tt1 = convert2weekly(TT1,'Aggregation',["lastvalue" "sum"]);   % Daily to weekly
tt2 = convert2weekly(TT ,'Aggregation',["lastvalue" "sum"]);   % Intra-daily to weekly

head(tt1)
```

```
       Time         Price     Log_Return
    ___________    ______    __________

    05-Jan-2018    112.78      0.12027
    12-Jan-2018    125.93      0.11029
    19-Jan-2018    117.67    -0.067842
    26-Jan-2018     118.8    0.0095573
    02-Feb-2018    120.85     0.017109
    09-Feb-2018    123.68     0.023147
    16-Feb-2018    124.33    0.0052417
    23-Feb-2018    127.09     0.021956
```

```
head(tt2)
```

```
       Time         Price     Log_Return
    ___________    ______    __________

    05-Jan-2018    112.78      0.12027
    12-Jan-2018    125.93      0.11029
    19-Jan-2018    117.67    -0.067842
    26-Jan-2018     118.8    0.0095573
    02-Feb-2018    120.85     0.017109
    09-Feb-2018    123.68     0.023147
    16-Feb-2018    124.33    0.0052417
    23-Feb-2018    127.09     0.021956
```

Notice that the results of the two approaches are the same and that convert2weekly reports on Fridays by default. For weeks in which Friday is not an NYSE trading day, the function reports results on the previous business day. In addition, you can use the convert2weekly optional name-value pair argument 'EndOfWeekDay' to specify a different day of the week that ends business weeks.

Input Arguments

TT1 — Data to aggregate to weekly periodicity
timetable

Data to aggregate to a weekly periodicity, specified as a timetable.

Each variable can be a numeric vector (univariate series) or numeric matrix (multivariate series).

Note

- NaNs indicate missing values.
- Timestamps must be in ascending or descending order.

By default, all days are business days. If your timetable does not account for nonbusiness days (weekends, holidays, and market closures), add business calendar awareness by using addBusinessCalendar first. For example, the following command adds business calendar logic to include only NYSE business days.

```
TT = addBusinessCalendar(TT);
```

Data Types: timetable

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: TT2 = convert2weekly(TT1,'Aggregation',["lastvalue" "sum"])

Aggregation — Aggregation method for TT1 data for intra-week or inter-day aggregation
"lastvalue" (default) | "sum" | "prod" | "mean" | "min" | "max" | "firstvalue" | character vector | function handle | string vector | cell vector of character vectors or function handles

Aggregation method for TT1 defining how data is aggregated over business days in an intra-week or inter-day periodicity, specified as one of the following methods, a string vector of methods, or a length numVariables cell vector of methods, where numVariables is the number of variables in TT1.

- "sum" — Sum the values in each year or day.
- "mean" — Calculate the mean of the values in each year or day.
- "prod" — Calculate the product of the values in each year or day.
- "min" — Calculate the minimum of the values in each year or day.
- "max" — Calculate the maximum of the values in each year or day.
- "firstvalue" — Use the first value in each year or day.
- "lastvalue" — Use the last value in each year or day.
- @customfcn — A custom aggregation method that accepts a table variable and returns a numeric scalar (for univariate series) or row vector (for multivariate series). The function must accept empty inputs [].

If you specify a single method, `convert2weekly` applies the specified method to all time series in `TT1`. If you specify a string vector or cell vector `aggregation`, `convert2weekly` applies `aggregation(j)` to `TT1(:,j)`; `convert2weekly` applies each aggregation method one at a time (for more details, see `retime`). For example, consider a daily timetable representing `TT1` with three variables.

```
  Time           AAA       BBB            CCC

___________    _______    _______    _________________
01-Jan-2018    100.00    200.00    300.00    400.00
02-Jan-2018    100.03    200.06    300.09    400.12
03-Jan-2018    100.07    200.14    300.21    400.28
04-Jan-2018    100.08    200.16    300.24    400.32
05-Jan-2018    100.25    200.50    300.75    401.00
06-Jan-2018    100.19    200.38    300.57    400.76
07-Jan-2018    100.54    201.08    301.62    402.16
08-Jan-2018    100.59    201.18    301.77    402.36
09-Jan-2018    101.40    202.80    304.20    405.60
10-Jan-2018    101.94    203.88    305.82    407.76
11-Jan-2018    102.53    205.06    307.59    410.12
12-Jan-2018    103.35    206.70    310.05    413.40
13-Jan-2018    103.40    206.80    310.20    413.60
14-Jan-2018    103.91    207.82    311.73    415.64
15-Jan-2018    103.89    207.78    311.67    415.56
16-Jan-2018    104.44    208.88    313.32    417.76
17-Jan-2018    104.44    208.88    313.32    417.76
18-Jan-2018    104.04    208.08    312.12    416.16
19-Jan-2018    104.94    209.88    314.82    419.76
```

The corresponding default weekly results representing `TT2` (in which all days are business days and the `'lastvalue'` is reported on Fridays) are as follows.

```
  Time         AAA       BBB            CCC

___________    _______    _______    _________________
05-Jan-2018    100.25    200.50    300.75    401.00
12-Jan-2018    103.35    206.70    310.05    413.40
19-Jan-2018    104.94    209.88    314.82    419.76
```

The default `'lastvalue'` returns the latest observed value in a given week for all variables in `TT1`.

All methods omit missing data (`NaN`s) in direct aggregation calculations on each variable. However, for situations in which missing values appear in the first row of `TT1`, missing values can also appear in the aggregated results `TT2`. To address missing data, write and specify a custom aggregation method (function handle) that supports missing data.

Data Types: `char` | `string` | `cell` | `function_handle`

`Daily` — Intra-day aggregation method for `TT1`
`"lastvalue"` (default) | `"sum"` | `"prod"` | `"mean"` | `"min"` | `"max"` | `"firstvalue"` | character vector | function handle | string vector | cell vector of character vectors or function handles

Intra-day aggregation method for `TT1`, specified as an aggregation method, a string vector of methods, or a length `numVariables` cell vector of methods. For more details on supported methods and behaviors, see the `'Aggregation'` name-value argument.

Data Types: `char` | `string` | `cell` | `function_handle`

EndOfWeekDay — Day of week that ends business weeks
"Friday" (weeks end on Friday) (default) | scalar integer with value 1 through 7 | "Sunday" | "Monday" | "Tuesday" | "Wednesday" | "Thursday" | "Friday" | "Saturday" | character vector

Day of the week that ends business weeks, specified as a value in the table.

Value	Day Ending Each Week
"Sunday" or 1	Sunday
"Monday" or 2	Monday
"Tuesday" or 3	Tuesday
"Wednesday" or 4	Wednesday
"Thursday" or 5	Thursday
"Friday" or 6	Friday
"Saturday" or 7	Saturday

If the specified end-of-week day in a given week is not a business day, the preceding business day ends that week.

Data Types: double | char | string

Output Arguments

TT2 — Weekly data
timetable

Weekly data, returned as a timetable. The time arrangement of TT1 and TT2 are the same.

If a variable of TT1 has no business-day records during an annual period within the sampling time span, convert2weekly returns a NaN for that variable and annual period in TT2.

If the first week (week1) of TT1 contains at least one business day, the first date in TT2 is the last business date of week1. Otherwise, the first date in TT2 is the next end-of-week business date of TT1.

If the last week (weekT) of TT1 contains at least one business day, the last date in TT2 is the last business date of weekT. Otherwise, the last date in TT2 is the previous end-of-week business date of TT1.

Version History

Introduced in R2021a

See Also

convert2daily | convert2quarterly | convert2semiannual | convert2monthly | convert2annual | timetable | addBusinessCalendar

Topics

“Resample and Aggregate Data in Timetable”
“Combine Timetables and Synchronize Their Data”
“Retime and Synchronize Timetable Variables Using Different Methods”

convert2monthly

Aggregate timetable data to monthly periodicity

Syntax

```
TT2 = convert2monthly(TT1)
TT2 = convert2monthly(TT1,Name,Value)
```

Description

`TT2 = convert2monthly(TT1)` aggregates data (for example, data recorded daily or weekly) to monthly periodicity.

`TT2 = convert2monthly(TT1,Name,Value)` uses additional options specified by one or more name-value arguments.

Examples

Aggregate Timetable Data to Monthly Periodicity

Apply separate aggregation methods to related variables in a `timetable` while maintaining consistency between aggregated results when converting to a monthly periodicity. You can use `convert2monthly` to aggregate both intra-daily data and aggregated daily data. These methods result in equivalent monthly aggregates. Lastly, you can aggregate results on a specific day of each month (for example, the 15th), rather than the default end of the month.

Load a timetable (TT) of simulated stock price data and corresponding logarithmic returns. The data stored in TT is recorded at various times throughout the day on New York Stock Exchange (NYSE) business days from January 1, 2018, to December 31, 2020. The timetable TT also includes NYSE business calendar awareness. If your timetable does not account for nonbusiness days (weekends, holidays, and market closures), add business calendar awareness by using `addBusinessCalendar` first.

```
load('SimulatedStock.mat','TT');
head(TT)
```

```
            Time            Price     Log_Return
    ____________________    ______    __________

    02-Jan-2018 11:52:11    100.71     0.0070749
    02-Jan-2018 13:23:09    103.11      0.023551
    02-Jan-2018 14:45:30    100.24     -0.028229
    02-Jan-2018 15:30:48    101.37       0.01121
    03-Jan-2018 10:02:21    101.81     0.0043311
    03-Jan-2018 11:22:37    100.17      -0.01624
    03-Jan-2018 14:45:20     99.66    -0.0051043
    03-Jan-2018 14:55:39    100.12     0.0046051
```

First aggregate intra-daily prices and returns to daily periodicity. To maintain consistency between prices and returns, for any given trading day aggregate prices by reporting the last recorded price using `"lastvalue"` and aggregate returns by summing all logarithmic returns using `"sum"`.

```
TT1 = convert2daily(TT,'Aggregation',["lastvalue" "sum"]);
head(TT1)
```

```
       Time         Price     Log_Return
    ___________    ______    __________

    02-Jan-2018    101.37     0.013607
    03-Jan-2018    100.12    -0.012408
    04-Jan-2018    106.76     0.064214
    05-Jan-2018    112.78     0.054856
    08-Jan-2018    119.07     0.054273
    09-Jan-2018    119.46      0.00327
    10-Jan-2018    124.44     0.040842
    11-Jan-2018    125.63    0.0095174
```

Use `convert2monthly` to aggregate the data to a monthly periodicity and compare the results of two different approaches. The first approach computes monthly results by aggregating the daily aggregates and the second approach computes monthly results by directly aggregating the original intra-daily data. Note that although `convert2monthly` reports results on the last business day of each month by default, you can report monthly results on the 15th of each month by using the optional name-value pair argument `'EndOfMonthDay'`.

```
tt1 = convert2monthly(TT1,'Aggregation',["lastvalue" "sum"],'EndOfMonthDay',15); % Daily to month
tt2 = convert2monthly(TT ,'Aggregation',["lastvalue" "sum"],'EndOfMonthDay',15); % Intra-daily to

head(tt1)
```

```
       Time         Price     Log_Return
    ___________    ______    __________

    12-Jan-2018    125.93      0.23056
    15-Feb-2018    120.55    -0.043662
    15-Mar-2018    113.49     -0.06035
    13-Apr-2018    112.07    -0.012591
    15-May-2018    110.47     -0.01438
    15-Jun-2018     99.06     -0.10902
    13-Jul-2018     95.74     -0.03409
    15-Aug-2018     99.94     0.042934
```

```
head(tt2)
```

```
       Time         Price     Log_Return
    ___________    ______    __________

    12-Jan-2018    125.93      0.23056
    15-Feb-2018    120.55    -0.043662
    15-Mar-2018    113.49     -0.06035
    13-Apr-2018    112.07    -0.012591
    15-May-2018    110.47     -0.01438
    15-Jun-2018     99.06     -0.10902
    13-Jul-2018     95.74     -0.03409
    15-Aug-2018     99.94     0.042934
```

Notice that the results of the two approaches are the same. For months in which the 15th is not an NYSE trading day, the function reports results on the previous business day.

Use Custom Aggregation Method to Convert Timetable Daily Data to Monthly Periodicity

You can apply custom aggregation methods using function handles. Specify a function handle to aggregate related variables in a `timetable` while maintaining consistency between aggregated results when converting from a daily to a monthly periodicity.

Load a timetable (TT) of simulated stock price data and corresponding logarithmic returns. The data stored in TT is recorded at various times throughout the day on New York Stock Exchange (NYSE) business days from January 1, 2018, to December 31,2020. The timetable TT also includes NYSE business calendar awareness. If your timetable does not account for nonbusiness days (weekends, holidays, and market closures), add business calendar awareness by using `addBusinessCalendar` first.

```
load('SimulatedStock.mat','TT')
head(TT)
```

```
            Time              Price     Log_Return
    ____________________    ______    __________

    02-Jan-2018 11:52:11    100.71     0.0070749
    02-Jan-2018 13:23:09    103.11      0.023551
    02-Jan-2018 14:45:30    100.24     -0.028229
    02-Jan-2018 15:30:48    101.37       0.01121
    03-Jan-2018 10:02:21    101.81     0.0043311
    03-Jan-2018 11:22:37    100.17      -0.01624
    03-Jan-2018 14:45:20     99.66    -0.0051043
    03-Jan-2018 14:55:39    100.12     0.0046051
```

First add another variable to TT that contains the simple (proportional) returns associated with the prices in TT and examine the first few rows.

```
TT.Simple_Return = exp(TT.Log_Return) - 1;  % Log returns to simple returns
head(TT)
```

```
            Time              Price     Log_Return    Simple_Return
    ____________________    ______    __________    _____________

    02-Jan-2018 11:52:11    100.71     0.0070749          0.0071
    02-Jan-2018 13:23:09    103.11      0.023551        0.023831
    02-Jan-2018 14:45:30    100.24     -0.028229       -0.027834
    02-Jan-2018 15:30:48    101.37       0.01121        0.011273
    03-Jan-2018 10:02:21    101.81     0.0043311       0.0043405
    03-Jan-2018 11:22:37    100.17      -0.01624       -0.016108
    03-Jan-2018 14:45:20     99.66    -0.0051043      -0.0050913
    03-Jan-2018 14:55:39    100.12     0.0046051       0.0046157
```

Create a function to aggregate simple returns and compute the monthly aggregates. To maintain consistency between prices and returns, for any given month, aggregate prices by reporting the last recorded price by using `"lastvalue"` and report logarithmic returns by summing all intervening logarithmic returns by using `"sum"`.

Notice that the aggregation function for simple returns operates along the first (row) dimension and omits missing data (NaNs). For more information on custom aggregation functions, see `timetable` and `retime`. When aggregation methods are a mix of supported methods and user-supplied functions, the `'Aggregation'` name-value pair argument must be specified as a cell vector of methods enclosed in curly braces.

```
f = @(x)(prod(1 + x,1,'omitnan') - 1);      % Aggregate simple returns
tt = convert2monthly(TT,'Aggregation',{'lastvalue' 'sum' f});
head(tt)
```

```
       Time         Price     Log_Return    Simple_Return
    ___________    ______    __________    _____________

    31-Jan-2018    122.96      0.20669            0.2296
    28-Feb-2018    121.92    -0.008494         -0.008458
    29-Mar-2018     108.9     -0.11294          -0.10679
    30-Apr-2018    110.38     0.013499           0.01359
    31-May-2018     99.02     -0.10861          -0.10292
    29-Jun-2018     96.24    -0.028477         -0.028075
    31-Jul-2018     97.15    0.0094111         0.0094555
    31-Aug-2018    101.51     0.043901          0.044879
```

Input Arguments

TT1 — Data to aggregate to monthly periodicity

timetable

Data to aggregate to a monthly periodicity, specified as a timetable.

Each variable can be a numeric vector (univariate series) or numeric matrix (multivariate series).

Note

- `NaN`s indicate missing values.
- Timestamps must be in ascending or descending order.

By default, all days are business days. If your timetable does not account for nonbusiness days (weekends, holidays, and market closures), add business calendar awareness by using `addBusinessCalendar` first. For example, the following command adds business calendar logic to include only NYSE business days.

```
TT = addBusinessCalendar(TT);
```

Data Types: `timetable`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `TT2 = convert2monthly(TT1,'Aggregation',["lastvalue" "sum"])`

Aggregation — Aggregation method for TT1

`"lastvalue"` (default) | `"sum"` | `"prod"` | `"mean"` | `"min"` | `"max"` | `"firstvalue"` | character vector | function handle | string vector | cell vector of character vectors or function handles

Aggregation method for TT1 defining how to aggregate data over business days in an intra-month or inter-day periodicity, specified as one of the following methods, a string vector of methods, or a length numVariables cell vector of methods, where numVariables is the number of variables in TT1.

- "sum" — Sum the values in each year or day.
- "mean" — Calculate the mean of the values in each year or day.
- "prod" — Calculate the product of the values in each year or day.
- "min" — Calculate the minimum of the values in each year or day.
- "max" — Calculate the maximum of the values in each year or day.
- "firstvalue" — Use the first value in each year or day.
- "lastvalue" — Use the last value in each year or day.
- @customfcn — A custom aggregation method that accepts a table variable and returns a numeric scalar (for univariate series) or row vector (for multivariate series). The function must accept empty inputs [].

If you specify a single method, convert2monthly applies the specified method to all time series in TT1. If you specify a string vector or cell vector aggregation, convert2monthly applies aggregation(*j*) to TT1(:,*j*); convert2monthly applies each aggregation method one at a time (for more details, see retime). For example, consider a daily timetable representing TT1 with three variables.

```
   Time         AAA       BBB           CCC
___________    ______    ______    ________________
01-Jan-2018    100.00    200.00    300.00    400.00
02-Jan-2018    100.03    200.06    300.09    400.12
03-Jan-2018    100.07    200.14    300.21    400.28
    .             .         .         .         .
    .             .         .         .         .
    .             .         .         .         .
31-Jan-2018    114.65     229.3    343.95    458.60
    .             .         .         .         .
    .             .         .         .         .
    .             .         .         .         .
28-Feb-2018    129.19    258.38    387.57    516.76
    .             .         .         .         .
    .             .         .         .         .
    .             .         .         .         .
31-Mar-2018    162.93    325.86    488.79    651.72
    .             .         .         .         .
    .             .         .         .         .
    .             .         .         .         .
30-Apr-2018    171.72    343.44    515.16    686.88
    .             .         .         .         .
    .             .         .         .         .
    .             .         .         .         .
31-May-2018    201.24    402.48    603.72    804.96
    .             .         .         .         .
    .             .         .         .         .
    .             .         .         .         .
30-Jun-2018    223.22    446.44    669.66    892.88
```

The corresponding default monthly results representing TT2 (in which all days are business days and the 'lastvalue' is reported on the last business day of each month) are as follows.

```
       Time         AAA       BBB          CCC      
    ___________    ______    ______    ________________

    31-Jan-2018    114.65    229.30    343.95    458.60
    28-Feb-2018    129.19    258.38    387.57    516.76
    31-Mar-2018    162.93    325.86    488.79    651.72
    30-Apr-2018    171.72    343.44    515.16    686.88
    31-May-2018    201.24    402.48    603.72    804.96
    30-Jun-2018    223.22    446.44    669.66    892.88
```

All methods omit missing data (NaNs) in direct aggregation calculations on each variable. However, for situations in which missing values appear in the first row of TT1, missing values can also appear in the aggregated results TT2. To address missing data, write and specify a custom aggregation method (function handle) that supports missing data.

Data Types: char | string | cell | function_handle

Daily — Intra-day aggregation method for TT1
"lastvalue" (default) | "sum" | "prod" | "mean" | "min" | "max" | "firstvalue" | character vector | function handle | string vector | cell vector of character vectors or function handles

Intra-day aggregation method for TT1, specified as an aggregation method, a string vector of methods, or a length numVariables cell vector of methods. For more details on supported methods and behaviors, see the 'Aggregation' name-value argument.

Data Types: char | string | cell | function_handle

EndOfMonthDay — Day of the month that ends months
last business day of month (default) | integer with value 1 to 31

Day of the month that ends months, specified as a scalar integer with value 1 to 31. For months with fewer days than EndOfMonthDay, convert2monthly reports aggregation results on the last business day of the month.

Data Types: double

Output Arguments

TT2 — Monthly data
timetable

Monthly data, returned as a timetable. The time arrangement of TT1 and TT2 are the same.

If a variable of TT1 has no business-day records during a month within the sampling time span, convert2monthly returns a NaN for that variable and month in TT2.

If the first month (month1) of TT1 contains at least one business day, the first date in TT2 is the last business date of month1. Otherwise, the first date in TT2 is the next end-of-month business date of TT1.

If the last month (monthT) of TT1 contains at least one business day, the last date in TT2 is the last business date of monthT. Otherwise, the last date in TT2 is the previous end-of-month business date of TT1.

Version History

Introduced in R2021a

See Also

convert2daily | convert2weekly | convert2quarterly | convert2semiannual | convert2annual | addBusinessCalendar | timetable

Topics

"Resample and Aggregate Data in Timetable"
"Combine Timetables and Synchronize Their Data"
"Retime and Synchronize Timetable Variables Using Different Methods"

convert2quarterly

Aggregate timetable data to quarterly periodicity

Syntax

```
TT2 = convert2quarterly(TT1)
TT2 = convert2quarterly(TT1,Name,Value)
```

Description

`TT2 = convert2quarterly(TT1)` aggregates data (for example, data recorded daily or weekly) to a quarterly periodicity.

`TT2 = convert2quarterly(TT1,Name,Value)` uses additional options specified by one or more name-value arguments.

Examples

Aggregate Timetable Data to Quarterly Periodicity

Apply separate aggregation methods to related variables in a `timetable` while maintaining consistency between aggregated results when converting to a quarterly periodicity. You can use `convert2quarterly` to aggregate both intra-daily data and aggregated monthly data. These methods result in equivalent quarterly aggregates.

Load a timetable (`TT`) of simulated stock price data and corresponding logarithmic returns. The data stored in `TT` is recorded at various times throughout the day on New York Stock Exchange (NYSE) business days from January 1, 2018, to December 31, 2020. The timetable `TT` also includes NYSE business calendar awareness. If your timetable does not account for nonbusiness days (weekends, holidays, and market closures), add business calendar awareness by using `addBusinessCalendar` first.

```
load('SimulatedStock.mat','TT');
head(TT)
```

```
            Time             Price     Log_Return
    ____________________    ______    __________

    02-Jan-2018 11:52:11    100.71     0.0070749
    02-Jan-2018 13:23:09    103.11      0.023551
    02-Jan-2018 14:45:30    100.24     -0.028229
    02-Jan-2018 15:30:48    101.37       0.01121
    03-Jan-2018 10:02:21    101.81     0.0043311
    03-Jan-2018 11:22:37    100.17      -0.01624
    03-Jan-2018 14:45:20     99.66    -0.0051043
    03-Jan-2018 14:55:39    100.12     0.0046051
```

Use `convert2monthly` to aggregate intra-daily prices and returns to a monthly periodicity. To maintain consistency between prices and returns for any given month, aggregate prices by reporting the last recorded price using "lastvalue" and aggregate returns by summing all logarithmic returns using "sum".

```
TT1 = convert2monthly(TT,'Aggregation',["lastvalue" "sum"]);
head(TT1)
```

```
       Time          Price     Log_Return
    ___________    ______    __________

    31-Jan-2018    122.96       0.20669
    28-Feb-2018    121.92     -0.008494
    29-Mar-2018     108.9      -0.11294
    30-Apr-2018    110.38      0.013499
    31-May-2018     99.02      -0.10861
    29-Jun-2018     96.24     -0.028477
    31-Jul-2018     97.15     0.0094111
    31-Aug-2018    101.51      0.043901
```

Use convert2quarterly to aggregate the data to a quarterly periodicity and compare the results of two different approaches. The first approach computes quarterly results by aggregating the monthly aggregates and the second approach computes quarterly results by directly aggregating the original intra-daily data. Note that convert2quaterly reports results on the last business day of each quarter.

```
tt1 = convert2quarterly(TT1,'Aggregation',["lastvalue" "sum"]);  % Monthly to quarterly
tt2 = convert2quarterly(TT ,'Aggregation',["lastvalue" "sum"]);  % Intra-daily to quarterly

head(tt1)
```

```
       Time          Price     Log_Return
    ___________    ______    __________

    29-Mar-2018     108.9       0.08526
    29-Jun-2018     96.24      -0.12358
    28-Sep-2018    111.37       0.14601
    31-Dec-2018     92.72      -0.18327
    29-Mar-2019      78.7      -0.16394
    28-Jun-2019    110.54       0.33973
    30-Sep-2019    180.13        0.4883
    31-Dec-2019    163.65     -0.095949
```

```
head(tt2)
```

```
       Time          Price     Log_Return
    ___________    ______    __________

    29-Mar-2018     108.9       0.08526
    29-Jun-2018     96.24      -0.12358
    28-Sep-2018    111.37       0.14601
    31-Dec-2018     92.72      -0.18327
    29-Mar-2019      78.7      -0.16394
    28-Jun-2019    110.54       0.33973
    30-Sep-2019    180.13        0.4883
    31-Dec-2019    163.65     -0.095949
```

The results of the two approaches are the same because each quarter contains exactly three calendar months.

Input Arguments

TT1 — Data to aggregate to quarterly periodicity
timetable

Data to aggregate to a quarterly periodicity, specified as a timetable.

Each variable can be a numeric vector (univariate series) or numeric matrix (multivariate series).

Note

- `NaN`s indicate missing values.
- Timestamps must be in ascending or descending order.

By default, all days are business days. If your timetable does not account for nonbusiness days (weekends, holidays, and market closures), add business calendar awareness by using `addBusinessCalendar` first. For example, the following command adds business calendar logic to include only NYSE business days.

```
TT = addBusinessCalendar(TT);
```

Data Types: `timetable`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `TT2 = convert2quarterly(TT1,'Aggregation',["lastvalue" "sum"])`

`Aggregation` — Aggregation method for TT1 data for intra-quarter or inter-day aggregation
`"lastvalue"` (default) | `"sum"` | `"prod"` | `"mean"` | `"min"` | `"max"` | `"firstvalue"` | character vector | function handle | string vector | cell vector of character vectors or function handles

Aggregation method for `TT1` data defining how to aggregate data over business days in an intra-quarter or inter-day periodicity, specified as one of the following methods, a string vector of methods, or a length `numVariables` cell vector of methods, where `numVariables` is the number of variables in `TT1`.

- `"sum"` — Sum the values in each year or day.
- `"mean"` — Calculate the mean of the values in each year or day.
- `"prod"` — Calculate the product of the values in each year or day.
- `"min"` — Calculate the minimum of the values in each year or day.
- `"max"` — Calculate the maximum of the values in each year or day.
- `"firstvalue"` — Use the first value in each year or day.
- `"lastvalue"` — Use the last value in each year or day.
- `@customfcn` — A custom aggregation method that accepts a table variable and returns a numeric scalar (for univariate series) or row vector (for multivariate series). The function must accept empty inputs `[]`.

If you specify a single method, `convert2quarterly` applies the specified method to all time series in `TT1`. If you specify a string vector or cell vector `aggregation`, `convert2quarterly` applies `aggregation(j)` to `TT1(:,j)`; `convert2quarterly` applies each aggregation method one at a time (for more details, see `retime`). For example, consider a daily timetable representing `TT1` with three variables.

```
   Time         AAA       BBB            CCC
___________    ______    ______    ________________
01-Jan-2018    100.00    200.00    300.00    400.00
02-Jan-2018    100.03    200.06    300.09    400.12
03-Jan-2018    100.07    200.14    300.21    400.28
    .             .         .         .         .
    .             .         .         .         .
    .             .         .         .         .
31-Mar-2018    162.93    325.86    488.79    651.72
    .             .         .         .         .
    .             .         .         .         .
    .             .         .         .         .
30-Jun-2018    223.22    446.44    669.66    892.88
    .             .         .         .         .
    .             .         .         .         .
    .             .         .         .         .
30-Sep-2018    232.17    464.34    696.51    928.68
    .             .         .         .         .
    .             .         .         .         .
    .             .         .         .         .
31-Dec-2018    243.17    486.34    729.51    972.68
```

The corresponding default quarterly results representing `TT2` (in which all days are business days and the `'lastvalue'` is reported on the last business day of each quarter) are as follows.

```
   Time         AAA       BBB            CCC
___________    ______    ______    ________________
31-Mar-2018    162.93    325.86    488.79    651.72
30-Jun-2018    223.22    446.44    669.66    892.88
30-Sep-2018    232.17    464.34    696.51    928.68
31-Dec-2018    243.17    486.34    729.51    972.68
```

All methods omit missing data (`NaN`s) in direct aggregation calculations on each variable. However, for situations in which missing values appear in the first row of `TT1`, missing values can also appear in the aggregated results `TT2`. To address missing data, write and specify a custom aggregation method (function handle) that supports missing data.

Data Types: `char` | `string` | `cell` | `function_handle`

`Daily` — Intra-day aggregation method for `TT1`
`"lastvalue"` (default) | `"sum"` | `"prod"` | `"mean"` | `"min"` | `"max"` | `"firstvalue"` | character vector | function handle | string vector | cell vector of character vectors or function handles

Intra-day aggregation method for `TT1`, specified as an aggregation method, a string vector of methods, or a length `numVariables` cell vector of methods. For more details on supported methods and behaviors, see the `'Aggregation'` name-value argument.

Data Types: `char` | `string` | `cell` | `function_handle`

Output Arguments

TT2 — Quarterly data
timetable

Quarterly data, returned as a timetable. The time arrangement of TT1 and TT2 are the same.

convert2quarterly reports quarterly aggregation results on the last business day of March, June, September, and December.

If a variable of TT1 has no business-day records during a quarter within the sampling time span, convert2quarterly returns a NaN for that variable and quarter in TT2.

If the first quarter (Q1) of TT1 contains at least one business day, the first date in TT2 is the last business date of Q1. Otherwise, the first date in TT2 is the next end-of-quarter business date of TT1.

If the last quarter (QT) of TT1 contains at least one business day, the last date in TT2 is the last business date of QT. Otherwise, the last date in TT2 is the previous end-of-quarter business date of TT1.

Version History

Introduced in R2021a

See Also

convert2daily | convert2weekly | convert2semiannual | convert2monthly | convert2annual | timetable | addBusinessCalendar

Topics

“Resample and Aggregate Data in Timetable”
“Combine Timetables and Synchronize Their Data”
“Retime and Synchronize Timetable Variables Using Different Methods”

convert2semiannual

Aggregate timetable data to semiannual periodicity

Syntax

```
TT2 = convert2semiannual(TT1)
TT2 = convert2semiannual(TT1,Name,Value)
```

Description

`TT2 = convert2semiannual(TT1)` aggregates data (for example, data recorded daily or weekly) to a semiannual periodicity.

`TT2 = convert2semiannual(TT1,Name,Value)` uses additional options specified by one or more name-value arguments.

Examples

Aggregate Timetable Data to Semiannual Periodicity

Apply separate aggregation methods to related variables in a `timetable` while maintaining consistency between aggregated results when converting to a semiannual periodicity. You can use `convert2semiannual` to aggregate both intra-daily data and aggregated quarterly data. These methods result in equivalent semiannual aggregates.

Load a timetable (`TT`) of simulated stock price data and corresponding logarithmic returns. The data stored in `TT` is recorded at various times throughout the day on New York Stock Exchange (NYSE) business days from January 1, 2018 to December 31,2020. The timetable `TT` also includes NYSE business calendar awareness. If your timetable does not account for nonbusiness days (weekends, holidays, and market closures), add business calendar awareness by using `addBusinessCalendar` first.

```
load('SimulatedStock.mat','TT');
head(TT)
```

```
            Time             Price     Log_Return
    ____________________    ______    __________

    02-Jan-2018 11:52:11    100.71     0.0070749
    02-Jan-2018 13:23:09    103.11      0.023551
    02-Jan-2018 14:45:30    100.24     -0.028229
    02-Jan-2018 15:30:48    101.37       0.01121
    03-Jan-2018 10:02:21    101.81     0.0043311
    03-Jan-2018 11:22:37    100.17      -0.01624
    03-Jan-2018 14:45:20     99.66    -0.0051043
    03-Jan-2018 14:55:39    100.12     0.0046051
```

Use `convert2quarterly` to aggregate intra-daily prices and returns to a quarterly periodicity. To maintain consistency between prices and returns, for any given quarter, aggregate prices by reporting the last recorded price using "lastvalue" and aggregate returns by summing all logarithmic returns using "sum".

```
TT1 = convert2quarterly(TT,'Aggregation',["lastvalue" "sum"])
```

```
TT1=12×2 timetable
       Time        Price     Log_Return
    ___________    ______    __________

    29-Mar-2018     108.9      0.08526 
    29-Jun-2018     96.24     -0.12358 
    28-Sep-2018    111.37      0.14601 
    31-Dec-2018     92.72     -0.18327 
    29-Mar-2019      78.7     -0.16394 
    28-Jun-2019    110.54      0.33973 
    30-Sep-2019    180.13       0.4883 
    31-Dec-2019    163.65    -0.095949 
    31-Mar-2020    177.46     0.081015 
    30-Jun-2020    168.96    -0.049083 
    30-Sep-2020    260.77      0.43398 
    31-Dec-2020    274.75     0.052223 
```

Use convert2semiannual to aggregate the data to a semiannual periodicity and compare the results of two different approaches. The first approach computes semiannual results by aggregating the quarterly aggregates and the second approach computes semiannual results by directly aggregating the original intra-daily data. Note that convert2semiannual reports results on the last business day of June and December.

```
tt1 = convert2semiannual(TT1,'Aggregation',["lastvalue" "sum"])  % Quarterly to semiannual
```

```
tt1=6×2 timetable
       Time        Price     Log_Return
    ___________    ______    __________

    29-Jun-2018     96.24    -0.038325 
    31-Dec-2018     92.72    -0.037261 
    28-Jun-2019    110.54      0.17579 
    31-Dec-2019    163.65      0.39235 
    30-Jun-2020    168.96     0.031932 
    31-Dec-2020    274.75       0.4862 
```

```
tt2 = convert2semiannual(TT ,'Aggregation',["lastvalue" "sum"])  % Intra-daily to semiannual
```

```
tt2=6×2 timetable
       Time        Price     Log_Return
    ___________    ______    __________

    29-Jun-2018     96.24    -0.038325 
    31-Dec-2018     92.72    -0.037261 
    28-Jun-2019    110.54      0.17579 
    31-Dec-2019    163.65      0.39235 
    30-Jun-2020    168.96     0.031932 
    31-Dec-2020    274.75       0.4862 
```

The results of the two approaches are the same because each semiannual period contains exactly two calendar quarters.

Input Arguments

TT1 — Data to aggregate to semiannual periodicity

timetable

Data to aggregate to a semiannual periodicity, specified as a timetable.

Each variable can be a numeric vector (univariate series) or numeric matrix (multivariate series).

Note

- NaNs indicate missing values.
- Timestamps must be in ascending or descending order.

By default, all days are business days. If your timetable does not account for nonbusiness days (weekends, holidays, and market closures), add business calendar awareness by using addBusinessCalendar first. For example, the following command adds business calendar logic to include only NYSE business days.

```
TT = addBusinessCalendar(TT);
```

Data Types: timetable

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: TT2 = convert2semiannual(TT1,'Aggregation',["lastvalue" "sum"])

Aggregation — Aggregation method for semiannual period to semiannual periodicity (inter-day aggregation)

"lastvalue" (default) | "sum" | "prod" | "mean" | "min" | "max" | "firstvalue" | character vector | function handle | string vector | cell vector of character vectors or function handles

Aggregation method for TT1 defining how data is aggregated over business days in a semiannual period to semiannual periodicity (inter-day aggregation), specified as one of the following methods, a string vector of methods, or a length numVariables cell vector of methods, where numVariables is the number of variables in TT1.

- "sum" — Sum the values in each year or day.
- "mean" — Calculate the mean of the values in each year or day.
- "prod" — Calculate the product of the values in each year or day.
- "min" — Calculate the minimum of the values in each year or day.
- "max" — Calculate the maximum of the values in each year or day.
- "firstvalue" — Use the first value in each year or day.
- "lastvalue" — Use the last value in each year or day.

- `@customfcn` — A custom aggregation method that accepts a table variable and returns a numeric scalar (for univariate series) or row vector (for multivariate series). The function must accept empty inputs `[]`.

If you specify a single method, `convert2semiannual` applies the specified method to all time series in `TT1`. If you specify a string vector or cell vector `aggregation`, `convert2semiannual` applies `aggregation(j)` to `TT1(:,j)`; `convert2semiannual` applies each aggregation method one at a time (for more details, see `retime`). For example, consider a daily timetable representing `TT1` with three variables.

```
    Time         AAA       BBB            CCC
___________    ______    ______    _________________
01-Jan-2018    100.00    200.00    300.00     400.00
02-Jan-2018    100.02    200.04    300.06     400.08
03-Jan-2018     99.96    199.92    299.88     399.84
    .             .         .         .          .
    .             .         .         .          .
    .             .         .         .          .
28-Jun-2018     69.63    139.26    208.89     278.52
29-Jun-2018     70.15     140.3    210.45     280.60
30-Jun-2018     75.77    151.54    227.31     303.08
01-Jul-2018     75.68    151.36    227.04     302.72
02-Jul-2018     71.34    142.68    214.02     285.36
03-Jul-2018     69.25    138.50    207.75     277.00
    .             .         .         .          .
    .             .         .         .          .
    .             .         .         .          .
29-Dec-2018    249.16    498.32    747.48     996.64
30-Dec-2018    250.21    500.42    750.63    1000.84
31-Dec-2018    256.75    513.50    770.25    1027.00
```

The corresponding default semiannual results representing `TT2` (in which all days are business days and the `'lastvalue'` is reported on the last business day of each semiannual period) are as follows.

```
     Time         AAA       BBB           CCC
___________    ______    ______    ________________
30-Jun-2018     75.77    151.54    227.31    303.08
31-Dec-2018    256.75    513.50    770.25   1027.00
```

All methods omit missing data (`NaN`s) in direct aggregation calculations on each variable. However, for situations in which missing values appear in the first row of `TT1`, missing values can also appear in the aggregated results `TT2`. To address missing data, write and specify a custom aggregation method (function handle) that supports missing data.

Data Types: `char` | `string` | `cell` | `function_handle`

Daily — Intra-day aggregation method for TT1

`"lastvalue"` (default) | `"sum"` | `"prod"` | `"mean"` | `"min"` | `"max"` | `"firstvalue"` | character vector | function handle | string vector | cell vector of character vectors or function handles

Intra-day aggregation method for `TT1`, specified as an aggregation method, a string vector of methods, or a length `numVariables` cell vector of methods. For more details on supported methods and behaviors, see the `'Aggregation'` name-value argument.

Data Types: `char` | `string` | `cell` | `function_handle`

Output Arguments

TT2 — Semiannual data
timetable

Semiannual data, returned as a timetable. `convert2semiannual` reports semiannual aggregation results on the last business day of June and December. The time arrangement of `TT1` and `TT2` are the same.

If a variable of `TT1` has no business-day records during an annual period within the sampling time span, `convert2semiannual` returns a `NaN` for that variable and annual period in `TT2`.

The first date in `TT2` is the last business date of the semiannual period in which the first date in `TT1` occurs, provided `TT1` has business dates in that semiannual period. Otherwise the first date in `TT2` is the next end-of-semiannual-period business date.

The last date in `TT2` is the last business date of the semiannual period in which the last date in `TT1` occurs, provided `TT1` has business dates in that semiannual period. Otherwise the last date in `TT2` is the previous end-of-semiannual-period business date.

Version History

Introduced in R2021a

See Also

`convert2daily` | `convert2weekly` | `convert2quarterly` | `convert2monthly` |
`convert2annual` | `timetable` | `addBusinessCalendar`

Topics

"Resample and Aggregate Data in Timetable"
"Combine Timetables and Synchronize Their Data"
"Retime and Synchronize Timetable Variables Using Different Methods"

convert2annual

Aggregate timetable data to annual periodicity

Syntax

```
TT2 = convert2annual(TT1)
TT2 = convert2annual(TT1,Name,Value)
```

Description

`TT2 = convert2annual(TT1)` aggregates data (for example, recorded daily or weekly data) to annual periodicity.

`TT2 = convert2annual(TT1,Name,Value)` uses additional options specified by one or more name-value arguments.

Examples

Aggregate Timetable Data to Annual Periodicity

Applyseparate aggregation methods to related variables in a `timetable` while maintaining consistency between aggregated results when converting to an annual periodicity. You can use `convert2annual` to aggregate both intra-daily data and aggregated monthly data. These methods result in equivalent annual aggregates.

Load a timetable (`TT`) of simulated stock price data and corresponding logarithmic returns. The data stored in `TT` is recorded at various times throughout the day on New York Stock Exchange (NYSE) business days from January 1, 2018, to December 31, 2020. The timetable `TT` also includes NYSE business calendar awareness. If your timetable does not account for nonbusiness days (weekends, holidays, and market closures), add business calendar awareness by using `addBusinessCalendar` first.

```
load('SimulatedStock.mat','TT');
head(TT)
```

```
            Time            Price     Log_Return
    ____________________    ______    __________

    02-Jan-2018 11:52:11    100.71     0.0070749
    02-Jan-2018 13:23:09    103.11      0.023551
    02-Jan-2018 14:45:30    100.24     -0.028229
    02-Jan-2018 15:30:48    101.37       0.01121
    03-Jan-2018 10:02:21    101.81     0.0043311
    03-Jan-2018 11:22:37    100.17      -0.01624
    03-Jan-2018 14:45:20     99.66    -0.0051043
    03-Jan-2018 14:55:39    100.12     0.0046051
```

First, aggregate intra-daily prices and returns to a monthly periodicity. To maintain consistency between prices and returns, for any given month aggregate prices by reporting the last recorded price using `"lastvalue"` and aggregate returns by summing all logarithmic returns using `"sum"`.

```
TT1 = convert2monthly(TT,'Aggregation',["lastvalue" "sum"]);
head(TT1)

       Time        Price     Log_Return
    ___________    ______    __________

    31-Jan-2018    122.96       0.20669
    28-Feb-2018    121.92     -0.008494
    29-Mar-2018     108.9      -0.11294
    30-Apr-2018    110.38      0.013499
    31-May-2018     99.02      -0.10861
    29-Jun-2018     96.24     -0.028477
    31-Jul-2018     97.15     0.0094111
    31-Aug-2018    101.51      0.043901
```

Use `convert2annual` to aggregate the data to an annual periodicity and compare the results of the two different aggregation approaches. The first approach computes annual results by aggregating the monthly aggregates and the second approach computes annual results by directly aggregating the original intra-daily data. Notice that by default, convert2annual reports results on the last business day of December. To change the month that ends annual periods, use the `'EndOfYearMonth'` name-value pair argument for `convert2annual`.

```
tt1 = convert2annual(TT1,'Aggregation',["lastvalue" "sum"])  % Monthly to annual

tt1=3×2 timetable
       Time        Price     Log_Return
    ___________    ______    __________

    31-Dec-2018     92.72     -0.075586
    31-Dec-2019    163.65       0.56815
    31-Dec-2020    274.75       0.51813
```

```
tt2 = convert2annual(TT ,'Aggregation',["lastvalue" "sum"])  % Intra-daily to semiannual

tt2=3×2 timetable
       Time        Price     Log_Return
    ___________    ______    __________

    31-Dec-2018     92.72     -0.075586
    31-Dec-2019    163.65       0.56815
    31-Dec-2020    274.75       0.51813
```

The results of the two approaches are the same because each annual period contains exactly 12 calendar months.

Input Arguments

TT1 — Data to aggregate to annual periodicity

timetable

Data to aggregate to an annual periodicity, specified as a timetable.

Each variable can be a numeric vector (univariate series) or numeric matrix (multivariate series).

Note

- NaNs indicate missing values.
- Timestamps must be in ascending or descending order.

By default, all days are business days. If your timetable does not account for nonbusiness days (weekends, holidays, and market closures), add business calendar awareness by using addBusinessCalendar first. For example, the following command adds business calendar logic to include only NYSE business days.

```
TT = addBusinessCalendar(TT);
```

Data Types: timetable

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: TT2 = convert2annual(TT1,'Aggregation',["lastvalue" "sum"])

Aggregation — Aggregation method for TT1
"lastvalue" (default) | "sum" | "prod" | "mean" | "min" | "max" | "firstvalue" | character vector | function handle | string vector | cell vector of character vectors or function handles

Aggregation method for TT1 defining how to aggregate data over business days in a year to an annual periodicity, specified as one of the following methods, a string vector of methods, or a length numVariables cell vector of methods, where numVariables is the number of variables in TT1.

- "sum" — Sum the values in each year or day.
- "mean" — Calculate the mean of the values in each year or day.
- "prod" — Calculate the product of the values in each year or day.
- "min" — Calculate the minimum of the values in each year or day.
- "max" — Calculate the maximum of the values in each year or day.
- "firstvalue" — Use the first value in each year or day.
- "lastvalue" — Use the last value in each year or day.
- @customfcn — A custom aggregation method that accepts a table variable and returns a numeric scalar (for univariate series) or row vector (for multivariate series). The function must accept empty inputs [].

If you specify a single method, convert2annual applies the specified method to all time series in TT1. If you specify a string vector or cell vector aggregation, convert2annual applies aggregation(*j*) to TT1(:,*j*); convert2annual applies each aggregation method one at a time (for more details, see retime). For example, consider an input daily timetable with three variables.

```
    Time         AAA       BBB           CCC
_____________   ______    ______    ________________
01-Jan-2018     100.00    200.00    300.00    400.00
02-Jan-2018     100.03    200.06    300.09    400.12
03-Jan-2018     100.07    200.14    300.21    400.28
```

```
          .            .         .         .         .
          .            .         .         .         .
          .            .         .         .         .
      29-Dec-2018    249.16    498.32    747.48    996.64
      30-Dec-2018    250.21    500.42    750.63   1000.84
      31-Dec-2018    256.75    513.50    770.25   1027.00
```

By default, `convert2annual` applies the aggregation method `"lastvalue"`, which reports for each variable the values of the last business day of each year. The aggregated annual results are as follows:

```
TT2 = convert2annual(TT1)
```

```
TT2 =

  1×3 timetable

        Time          AAA       BBB             CCC
    ___________    ______    ______    ________________
    31-Dec-2018    256.75    513.50    770.25   1027.00
```

All methods omit missing data (`NaN`s) in direct aggregation calculations on each variable. However, for situations in which missing values appear in the first row of `TT1`, missing values can also appear in the aggregated results `TT2`. To address missing data, write and specify a custom aggregation method (function handle) that supports missing data.

Data Types: `char` | `string` | `cell` | `function_handle`

Daily — Intra-day aggregation method for TT1

`"lastvalue"` (default) | `"sum"` | `"prod"` | `"mean"` | `"min"` | `"max"` | `"firstvalue"` | character vector | function handle | string vector | cell vector of character vectors or function handles

Intra-day aggregation method for `TT1`, specified as an aggregation method, a string vector of methods, or a length `numVariables` cell vector of methods. For more details on supported methods and behaviors, see the `'Aggregation'` name-value argument.

Data Types: `char` | `string` | `cell` | `function_handle`

EndOfYearMonth — Month that ends annual periods

`"December"` (weeks end on Friday) (default) | integer with value `1` to `12` | `"January"` | `"February"` | `"March"` | `"April"` | `"May"` | `"June"` | `"July"` | `"August"` | `"September"` | `"October"` | `"November"` | `"December"` | character vector

Month that ends annual periods, specified as a value in this table.

Value	Month Ending Each Year
`"January"` or `1`	January
`"February"` or `2`	February
`"March"` or `3`	March
`"April"` or `4`	April
`"May"` or `5`	May
`"June"` or `6`	June
`"July"` or `7`	July

Value	Month Ending Each Year
`"August"` or `8`	August
`"September"` or `9`	September
`"October"` or `10`	October
`"November"` or `11`	November
`"December"` or `12`	December

Data Types: `double` | `char` | `string`

Output Arguments

TT2 — Annual data
timetable

Annual data, returned as a timetable. The time arrangement of `TT1` and `TT2` are the same.

If a variable of `TT1` has no business-day records during an annual period within the sampling time span, `convert2annual` returns a `NaN` for that variable and annual period in `TT2`.

If the first annual period (`year1`) of `TT1` contains at least one business day, the first date in `TT2` is the last business date of `year1`. Otherwise, the first date in `TT2` is the next end-of-year-period business date of `TT1`.

If the last annual period (`yearT`) of `TT1` contains at least one business day, the last date in `TT2` is the last business date of `yearT`. Otherwise, the last date in `TT2` is the previous end-of-year-period business date of `TT1`.

Version History

Introduced in R2021a

See Also

`convert2daily` | `convert2weekly` | `convert2quarterly` | `convert2semiannual` | `convert2monthly` | `timetable` | `addBusinessCalendar`

Topics

“Resample and Aggregate Data in Timetable”
“Combine Timetables and Synchronize Their Data”
“Retime and Synchronize Timetable Variables Using Different Methods”

totalreturnprice

Total return price time series

Syntax

```
Return = totalreturnprice(Price,Action,Dividend)
```

Description

`Return = totalreturnprice(Price,Action,Dividend)` generates a total return price time series given price data, action or split data, and dividend data. All input data is unadjusted.

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=2×2 table
        Date        Return
    ___________    ______

    01-Jan-2005         1
    02-Jan-2005    1.0833
```

Input Arguments

Price — Price of security

matrix | table

Price of security, specified as a table or an `NUMOBS`-by-2 matrix. If `Price` is a table, the dates can either be a datetime array, string array, date character vectors, or serial date numbers. If `Price` is an `NUMOBS`-by-2 matrix of price data, column 1 contains MATLAB serial date numbers and column 2 contains price values.

Data Types: `double` | `char` | `string` | `datetime` | `table`

Action — Action or split data

matrix | table

Action or split data, specified as a table or an NUMOBS-by-2 matrix. If Action is a table, the dates can either be a datetime array, string array, date character vectors, or serial date numbers. If Action is an NUMOBS-by-2 matrix of price data, column 1 contains MATLAB serial date numbers and column 2 contains split ratios.

Data Types: double | char | string | datetime | table

Dividend — Dividend payouts

matrix | table

Dividend payouts, specified as a table or an NUMOBS-by-2 matrix. If Dividend is a table, the dates can either be serial date numbers, date character vectors, or datetime arrays. If Dividend is a NUMOBS-by-2 matrix of price data, column 1 contains MATLAB serial date numbers and column 2 contains dividend payouts.

Data Types: double | table

Output Arguments

Return — Total return price time series

matrix | table

Total return price time series, returned as a NUMOBS-by-2 matrix (if all inputs are matrices) or table (if any inputs are tables) of price data, where column 1 is dates and column 2 is total return price values. Dates in column 1 are in datetime format if any inputs specify dates in datetime format. Dates in column 1 are in date character vector format if no inputs specify dates in datetime format, but any of them use date character vector format. Otherwise, dates in column 1 are specified as serial date numbers. Use the function datestr to convert serial date numbers to formatted date character vectors.

Version History

Introduced before R2006a

See Also

periodicreturns | datetime

Topics

“Portfolio Construction Examples” on page 3-5
“Portfolio Optimization Functions” on page 3-3

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 = 4×6
10^5 ×

    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 = 4×1

  100.1562
  100.3750
   99.5000
  101.0938


Yields = 4×1

    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(1997,1,1))
```

```
Bonds=4×6 table
     Maturity      CouponRate     Face     Period     Basis     EndMonthRule
    ___________    __________     ____     ______     _____     ____________

    26-Mar-1998     0.06125       100        2          0            1
    30-Jul-1998      0.0625       100        2          0            1
    17-Dec-1998     0.05125       100        2          0            1
    15-Apr-1999       0.065       100        2          0            1


Prices = 4×1

  100.1562
  100.3750
   99.5000
  101.0938


Yields = 4×1

    0.0598
    0.0599
    0.0540
    0.0598
```

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 a datetime array, string array, or date character vectors.
- 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 | char | string | datetime | table

Settle — Settlement date of Treasury bond

datetime array | string array | date character vector

(Optional) Settlement date of the Treasury bond, specified as a scalar or a NINST-by-1 vector using a datetime array, string array, or date character vectors. The Settle date must be before the Maturity date.

To support existing code, tr2bonds also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

Output Arguments

Bonds — Coupon bond information

table | matrix

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. The format of the dates matches the format used for Maturity in TreasuryMatrix (datetime array, string array, date character vector, or serial date number).
- 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 2-16.
- `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.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although `tr2bonds` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)

y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

tbl2bond | zbtprice | zbtyield | datetime

Topics

“Term Structure of Interest Rates” on page 2-29
“Treasury Bills Defined” on page 2-25

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=10×3 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 = 8×8

   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
```

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 = 8×8

   93.1345    5.9335    0.7456    0.1553    0.0311         0         0         0
    1.7359   92.9198    4.5446    0.6046    0.1560         0         0    0.0390
    0.1268    2.9716   91.9913    4.3124    0.4711    0.0544         0    0.0725
    0.0210    0.3785    5.0683   89.7792    4.0379    0.4627    0.0421    0.2103
    0.0221    0.1105    0.6851    6.2320   88.3757    3.6464    0.2873    0.6409
         0         0    0.0761    0.7230    7.9909   86.1872    2.7397    2.2831
         0         0         0    0.3094    1.8561    4.5630   80.8971   12.3743
         0         0         0         0         0         0         0  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=10×3 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 = 3×3

   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=10×3 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 = 3×3

   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 = 8×8

   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
```

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 = 8×8
```

```
   93.1345    5.9335    0.7456    0.1553    0.0311         0         0         0
    1.7359   92.9198    4.5446    0.6046    0.1560         0         0    0.0390
    0.1268    2.9716   91.9913    4.3124    0.4711    0.0544         0    0.0725
    0.0210    0.3785    5.0683   89.7792    4.0379    0.4627    0.0421    0.2103
    0.0221    0.1105    0.6851    6.2320   88.3757    3.6464    0.2873    0.6409
         0         0    0.0761    0.7230    7.9909   86.1872    2.7397    2.2831
         0         0         0    0.3094    1.8561    4.5630   80.8971   12.3743
         0         0         0         0         0         0         0  100.0000
```

Visualize Transitions Data for `transprob`

This example shows how to visualize credit rating transitions that are used as an input to the `transprob` function. The example also describes how the `transprob` function treats rating transitions when the company data starts after the start date of the analysis, or when the end date of the analysis is after the last transition observed.

Sample Data

Set up fictitious sample data for illustration purposes.

```
data = {'ABC','17-Feb-2015','AA';
    'ABC','6-Jul-2017','A';
    'LMN','12-Aug-2014','B';
    'LMN','9-Nov-2015','CCC';
    'LMN','7-Sep-2016','D';
    'XYZ','14-May-2013','BB';
    'XYZ','21-Jun-2016','BBB'};
data = cell2table(data,'VariableNames',{'ID','Date','Rating'});
disp(data)
```

```
       ID              Date            Rating
    _______    ________________    _______

    {'ABC'}    {'17-Feb-2015'}    {'AA' }
    {'ABC'}    {'6-Jul-2017' }    {'A'  }
    {'LMN'}    {'12-Aug-2014'}    {'B'  }
    {'LMN'}    {'9-Nov-2015' }    {'CCC'}
    {'LMN'}    {'7-Sep-2016' }    {'D'  }
    {'XYZ'}    {'14-May-2013'}    {'BB' }
    {'XYZ'}    {'21-Jun-2016'}    {'BBB'}
```

The `transprob` function understands that this panel-data format indicates the dates when a new rating is assigned to a given company. `transprob` assumes that such ratings remain unchanged, unless a subsequent row explicitly indicates a rating change. For example, for company `'ABC'`, `transprob` understands that the `'A'` rating is unchanged for any date after `'6-Jul-2017'` (indefinitely).

Compute Transition Matrix and Transition Counts

The `transprob` function returns a transition probability matrix as the primary output. There are also optional outputs that contain additional information for how many transitions occurred. For more information, see `transprob` for information on the optional outputs for both the `'cohort'` and the `'duration'` methods.

For illustration purposes, this example allows you to pick the `StartYear` (limited to `2014` or `2015` for this example) and the `EndYear` (`2016` or `2017`). This example also uses the `hDisplayTransitions` helper function (see the Local Functions on page 15-1534 section) to format the transitions information for ease of reading.

```
StartYear =  2014 ;

EndYear =  2017 ;
startDate = datetime(StartYear,12,31,'Locale','en_US');
endDate = datetime(EndYear,12,31,'Locale','en_US');
RatingLabels = ["AAA","AA","A","BBB","BB","B","CCC","D"];

[tm,st,it] = transprob(data,'startDate',startDate,'endDate',endDate,'algorithm','cohort','labels
```

The transition probabilities of the `TransMat` output indicate the probability of migrating between ratings. The probabilities are expressed in %, that is, they are multiplied by 100.

```
hDisplayTransitions(tm,RatingLabels,"Transition Matrix")
```

```
Transition Matrix

           AAA    AA     A     BBB    BB    B    CCC     D 
           ___    __    ___    ___    __    _    ___    ___

    AAA    100     0      0      0     0    0      0      0
    AA       0    50     50      0     0    0      0      0
    A        0     0    100      0     0    0      0      0
    BBB      0     0      0    100     0    0      0      0
    BB       0     0      0     50    50    0      0      0
    B        0     0      0      0     0    0    100      0
    CCC      0     0      0      0     0    0      0    100
    D        0     0      0      0     0    0      0    100
```

The transition counts are stored in the `sampleTotals` optional output and indicate how many transitions occurred between ratings for the entire sample (that is, all companies).

```
hDisplayTransitions(st.totalsMat,RatingLabels,"Transition counts, all companies")
```

```
Transition counts, all companies

           AAA    AA    A    BBB    BB    B    CCC    D
           ___    __    _    ___    __    _    ___    _

    AAA     0     0     0     0     0     0     0     0
    AA      0     1     1     0     0     0     0     0
    A       0     0     0     0     0     0     0     0
    BBB     0     0     0     1     0     0     0     0
    BB      0     0     0     1     1     0     0     0
    B       0     0     0     0     0     0     1     0
    CCC     0     0     0     0     0     0     0     1
    D       0     0     0     0     0     0     0     1
```

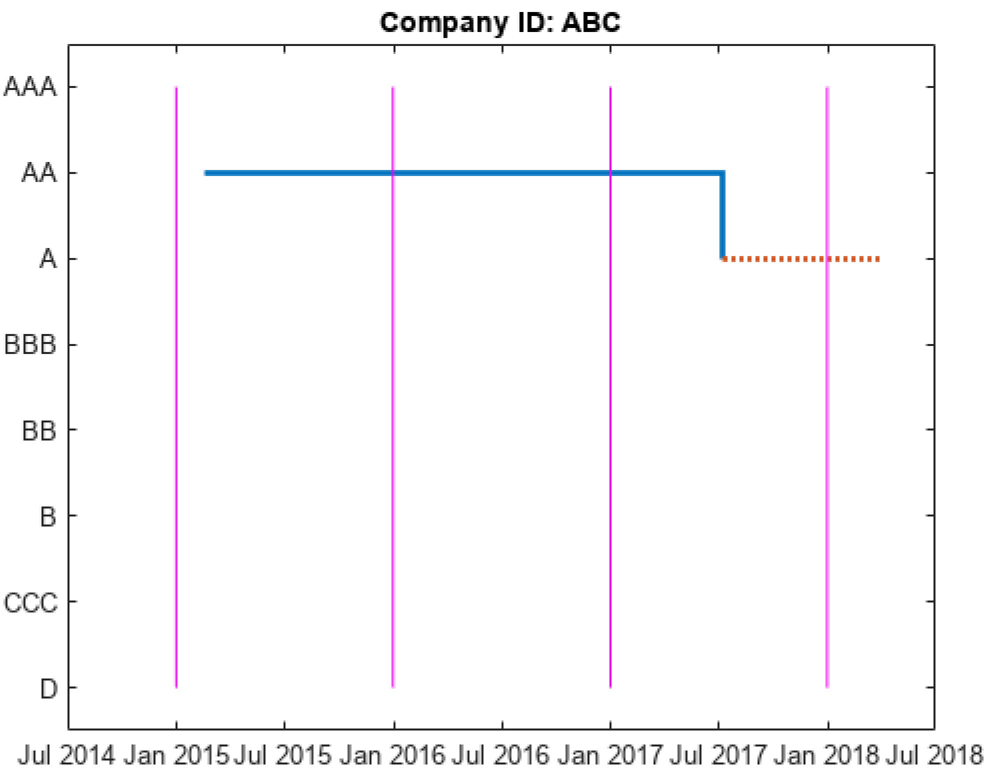
The third output of `transprob` is `idTotals` that contains information about transitions at an ID level, company by company (in the same order that the companies appear in the input data).

Select a company to display the transition counts and a corresponding visualization of the transitions. The `hPlotTransitions` helper function (see the Local Functions on page 15-1534 section) shows the transitions history for a company.

```
CompanyID = ABC ;
UniqueIDs = unique(data.ID,'stable');
[~,CompanyIndex] = ismember(CompanyID,UniqueIDs);
hDisplayTransitions(it(CompanyIndex).totalsMat,RatingLabels,strcat("Transition counts, company I
```

```
Transition counts, company ID: ABC

           AAA    AA    A    BBB    BB    B    CCC    D
           ___    __    _    ___    __    _    ___    _

    AAA     0     0     0     0     0     0     0     0
    AA      0     1     1     0     0     0     0     0
    A       0     0     0     0     0     0     0     0
    BBB     0     0     0     0     0     0     0     0
    BB      0     0     0     0     0     0     0     0
    B       0     0     0     0     0     0     0     0
    CCC     0     0     0     0     0     0     0     0
    D       0     0     0     0     0     0     0     0
```

```
hPlotTransitions(CompanyID,startDate,endDate,data,RatingLabels)
```



To understand how `transprob` handles data when the first observed date is after the start date of the analysis, or whose last observed date occurs before the end date of the analysis, consider the following example. For company `'ABC'` suppose that the analysis has a start date of `31-Dec-2014` and end date of `31-Dec-2017`. There are only two transitions reported for this company for that analysis time window. The first observation for `'ABC'` happened on `17-Feb-2015`. So the `31-Dec-2015` snapshot is the first time the company is observed. By `31-Dec-2016`, the company remained in the original 'AA' rating. By 31-Dec-2017, a downgrade to 'A' is recorded. Consistent with this, the transition counts show one transition from 'AA' to 'AA' (from the end of 2015 to the end of 2016), and one transition from 'AA' to 'A' (from the end of 2016 to the end of 2017). The plot shows the last rating as a dotted red line to emphasize that the last rating in the data is extrapolated indefinitely into the future. There is no extrapolation into the past; the company's history is ignored until a company rating is known for an entire transition period (31-Dec-2015 through 31-Dec-2016 in the case of 'ABC').

Compute Transition Matrix Containing NR (Not Rated) Rating

Consider a different sample data containing only a single company 'DEF'. The data contains transitions of company 'DEF' from 'A' to 'NR' rating and a subsequent transition from 'NR' to 'BBB'.

```
dataNR = {'DEF','17-Mar-2011','A';
    'DEF','24-Mar-2014','NR';
    'DEF','26-Sep-2016','BBB'};
dataNR = cell2table(dataNR,'VariableNames',{'ID','Date','Rating'});
disp(dataNR)
```

```
      ID            Date            Rating
    _______    _______________    _______

    {'DEF'}    {'17-Mar-2011'}    {'A'  }
    {'DEF'}    {'24-Mar-2014'}    {'NR' }
    {'DEF'}    {'26-Sep-2016'}    {'BBB'}
```

transprob treats 'NR' as another rating. The transition matrix below shows the estimated probability of transitioning into and out of 'NR'.

```
StartYearNR = 2010;
EndYearNR = 2018;
startDateNR = datetime(StartYearNR,12,31,'Locale','en_US');
endDateNR = datetime(EndYearNR,12,31,'Locale','en_US');
CompanyID_NR = "DEF";

RatingLabelsNR = ["AAA","AA","A","BBB","BB","B","CCC","D","NR"];

[tmNR,~,itNR] = transprob(dataNR,'startDate',startDateNR,'endDate',endDateNR,'algorithm','cohort
hDisplayTransitions(tmNR,RatingLabelsNR,"Transition Matrix")
```

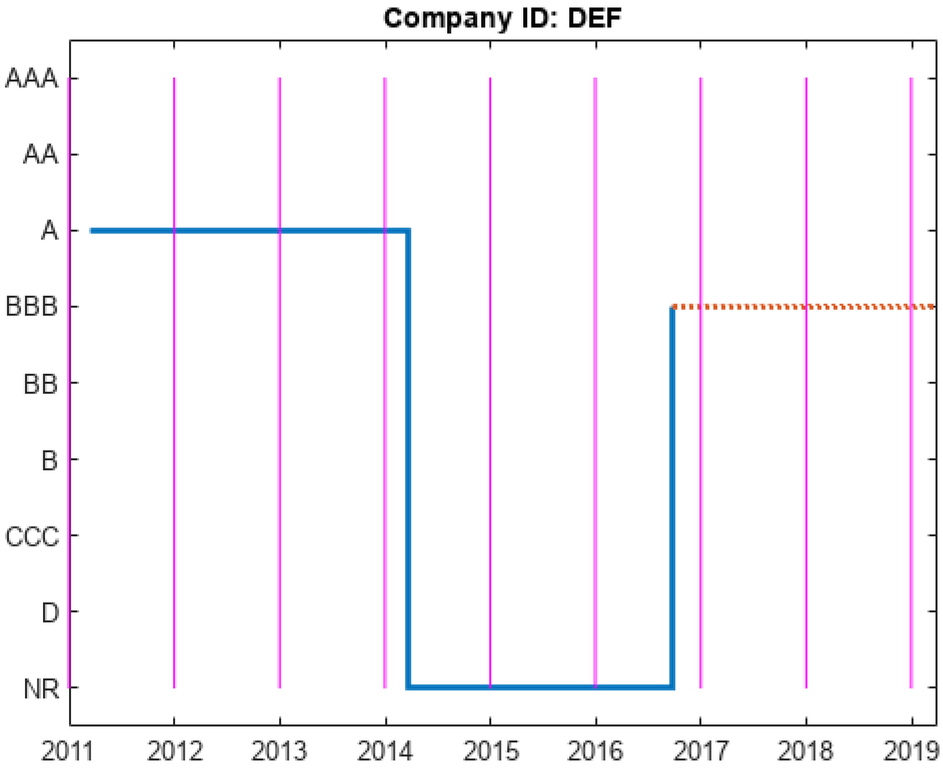
```
Transition Matrix

           AAA    AA       A       BBB    BB      B     CCC     D       NR  
           ___    ___    ______    ___    ___    ___    ___    ___    ______

    AAA    100      0         0      0      0      0      0      0         0
    AA       0    100         0      0      0      0      0      0         0
    A        0      0    66.667      0      0      0      0      0    33.333
    BBB      0      0         0    100      0      0      0      0         0
    BB       0      0         0      0    100      0      0      0         0
    B        0      0         0      0      0    100      0      0         0
    CCC      0      0         0      0      0      0    100      0         0
    D        0      0         0      0      0      0      0    100         0
    NR       0      0         0     50      0      0      0      0        50
```

Display the transition counts and corresponding visualization of the transitions.

```
hDisplayTransitions(itNR.totalsMat,RatingLabelsNR,strcat("Transition counts, company ID: ",Compa
```

```
Transition counts, company ID: DEF

           AAA    AA    A    BBB    BB    B    CCC    D    NR
           ___    __    _    ___    __    _    ___    _    __

    AAA     0     0     0     0     0     0     0     0     0
    AA      0     0     0     0     0     0     0     0     0
    A       0     0     2     0     0     0     0     0     1
    BBB     0     0     0     2     0     0     0     0     0
    BB      0     0     0     0     0     0     0     0     0
    B       0     0     0     0     0     0     0     0     0
    CCC     0     0     0     0     0     0     0     0     0
    D       0     0     0     0     0     0     0     0     0
    NR      0     0     0     1     0     0     0     0     1
```

```
hPlotTransitions(CompanyID_NR,startDateNR,endDateNR,dataNR,RatingLabelsNR)
```



To remove the `'NR'` from the transition matrix, use the `'excludeLabels'` name-value input argument in `transprob`. The list of labels to exclude may or may not be specified in the name-value pair argument `labels`. For example, both `RatingLabels` and `RatingLabelsNR` generate the same output from `transprob`.

```
[tmNR,stNR,itNR] = transprob(dataNR,'startDate',startDateNR,'endDate',endDateNR,'algorithm','coh
hDisplayTransitions(tmNR,RatingLabels,"Transition Matrix")
```

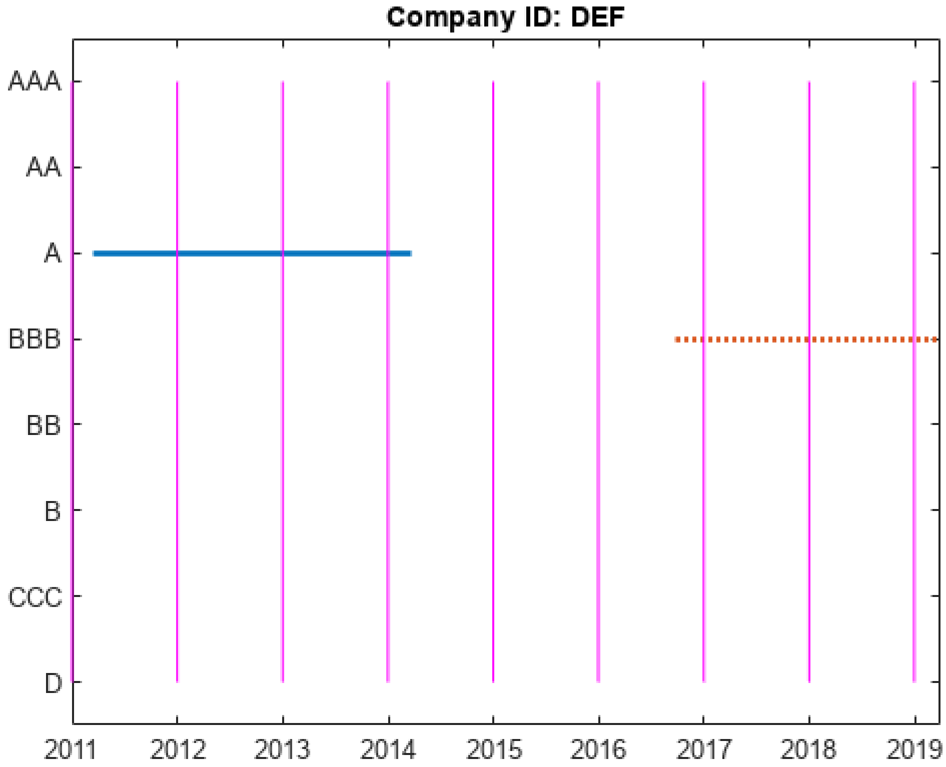
```
Transition Matrix

           AAA    AA      A      BBB    BB      B      CCC     D
           ___    ___    ___    ___    ___    ___    ___    ___

    AAA    100      0      0      0      0      0      0      0
    AA       0    100      0      0      0      0      0      0
    A        0      0    100      0      0      0      0      0
    BBB      0      0      0    100      0      0      0      0
    BB       0      0      0      0    100      0      0      0
    B        0      0      0      0      0    100      0      0
    CCC      0      0      0      0      0      0    100      0
    D        0      0      0      0      0      0      0    100
```

Display the transition counts and corresponding visualization of the transitions.

```
hDisplayTransitions(itNR.totalsMat,RatingLabels,strcat("Transition counts, company ID: ",CompanyI
```

```
Transition counts, company ID: DEF

           AAA    AA    A    BBB    BB    B    CCC    D
           ___    __    _    ___    __    _    ___    _

    AAA     0     0     0     0     0     0     0     0
    AA      0     0     0     0     0     0     0     0
    A       0     0     2     0     0     0     0     0
    BBB     0     0     0     2     0     0     0     0
    BB      0     0     0     0     0     0     0     0
    B       0     0     0     0     0     0     0     0
    CCC     0     0     0     0     0     0     0     0
    D       0     0     0     0     0     0     0     0
```

```
hPlotTransitions(CompanyID_NR,startDateNR,endDateNR,dataNR,RatingLabels)
```



Consistent with the previous plot, the transition counts still show two transitions from `'A'` to `'A'` (from the end of 2012 to the end of 2014), and two transitions from `'BBB'` to `'BBB'` (from the end of 2017 to the end of 2019).

However, different from the previous plot, specifying `'NR'` using the `'excludeLabels'` name-value input argument of `transprob` removes any transitions into and out of the `'NR'` rating.

Local Functions

```
function hDisplayTransitions(TransitionsData,RatingLabels,Title)
% Helper function to format transition information outputs

TransitionsAsTable = array2table(TransitionsData,...
   'VariableNames',RatingLabels,'RowNames',RatingLabels);

fprintf('\n%s\n\n',Title)
disp(TransitionsAsTable)

end

function hPlotTransitions(CompanyID,startDate,endDate,data,RatingLabels)
% Helper function to visualize transitions between ratings

   Ind = string(data.ID)==CompanyID;
   DatesOriginal = datetime(data.Date(Ind),'Locale','en_US');
   RatingsOriginal = categorical(data.Rating(Ind),flipud(RatingLabels(:)),flipud(RatingLabels(:)
   stairs(DatesOriginal,RatingsOriginal,'LineWidth',2)
   hold on;

   % Indicate rating extrapolated into the future (arbitrarily select 91
   % days after endDate as the last date on the plot)
   endDateExtrap = endDate+91;
   if endDateExtrap>DatesOriginal(end)
      DatesExtrap = [DatesOriginal(end); endDateExtrap];
      RatingsExtrap = [RatingsOriginal(end); RatingsOriginal(end)];
      stairs(DatesExtrap,RatingsExtrap,'LineWidth',2,'LineStyle',':')
   end
   hold off;

   % Add lines to indicate the snapshot dates
   % transprob uses 1 as the default for 'snapsPerYear', hardcoded here for simplicity
   % The call to cfdates generates the exact same snapshot dates that transprob uses
   snapsPerYear = 1;
   snapDates = cfdates(startDate-1,endDate,snapsPerYear)';
   yLimits = ylim;
   for ii=1:length(snapDates)
      line([snapDates(ii) snapDates(ii)],yLimits,'Color','m')
   end
   title(strcat("Company ID: ",CompanyID))
end
```

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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `transMat = transprob(data,'algorithm','cohort')`

`algorithm` — Estimation algorithm

`'duration'` (default) | character vector with values are `'duration'` or `'cohort'`

Estimation algorithm, specified as the comma-separated pair consisting of `'algorithm'` and 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) | datetime array | string array | date character vector | serial date number

End date of the estimation time window, specified as the comma-separated pair consisting of 'endDate' and a scalar datetime, string, date character vector, or serial date number. The endDate cannot be a date before the startDate.

Data Types: char | double | string | datetime

labels — Credit-rating scale

{'AAA','AA','A','BBB','BB','B','CCC','D'} (default) | cell array of character vectors

Credit-rating scale, specified as the comma-separated pair consisting of 'labels' and 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 the comma-separated pair consisting of 'snapsPerYear' and 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) | datetime array | string array | date character vector | serial date number

Start date of the estimation time window, specified as the comma-separated pair consisting of 'startDate' and a scalar datetime, string, date character vector, or serial date number.

Data Types: char | double | string | datetime

transInterval — Length of the transition interval in years

1 (one year transition probability) (default) | numeric

Length of the transition interval, in years, specified as the comma-separated pair consisting of 'transInterval' and a numeric value.

Data Types: double

excludeLabels — Label that is excluded from the transition probability computation

'' (do not exclude any label) (default) | numeric | character vector | string

Label that is excluded from the transition probability computation, specified as the comma-separated pair consisting of 'excludeLabels' and a character vector, string, or numerical rating.

If multiple labels are to be excluded, 'excludeLabels' must be a cell array containing all of the labels for exclusion. The type of the labels given in 'excludeLabels' must be consistent with the data type specified in the labels input.

The list of labels to exclude may or may not be specified in labels.

Data Types: double | char | string

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.

More About

Cohort Estimation

The cohort algorithm estimates the transition probabilities based on a sequence of snapshots of credit ratings at regularly spaced points in time.

If the credit rating of a company changes twice between two snapshot dates, the intermediate rating is overlooked and only the initial and final ratings influence the estimates.

Duration Estimation

Unlike the cohort method, the duration algorithm estimates the transition probabilities based on the full credit ratings history, looking at the exact dates on which the credit rating migrations occur.

There is no concept of snapshots in this method, and all credit rating migrations influence the estimates, even when a company's rating changes twice within a short time.

Algorithms

Cohort Estimation

The algorithm first determines a sequence $t_0,...,t_K$ 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 byN_{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 byP_{ij}, is

$$P_{ij} = \frac{Nij}{Ni}$$

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, Δ*t*, 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 $\Delta t = 2$, *P* contains the two-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.

When ratings must be excluded (see the `excludeLabels` name-value input argument), all transitions involving the excluded ratings are removed from the sample. For example, if the `‘NR’` rating must be excluded, any transitions into `‘NR’` and out of `‘NR’` are excluded from the sample. The total counts for all other ratings are adjusted accordingly. For more information, see “Visualize Transitions Data for transprob” on page 8-111.

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

$$\lambda_{ij} = \frac{T_{ij}}{T_i}, i \neq j$$

The diagonal entries are computed as:

$$\lambda_{ii} = -\sum_{j \neq i} \lambda_{ij}$$

With the generator matrix and the transition interval Δt (e.g., $\Delta t = 2$ corresponds to two-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.

When ratings must be excluded (see the `excludeLabels` name-value input argument), all transitions involving the exclude ratings are removed from the sample. For example, if the `‘NR’` rating must be excluded, any transitions into `‘NR’` and out of `‘NR’` are excluded from the sample. The total time spent in `‘NR’` (or any other excluded rating) is also removed.

Version History

Introduced in R2010b

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

`transprobbytotals` | `transprobprep` | `table`

Topics

"Estimation of Transition Probabilities" on page 8-2
"Estimate Transition Probabilities for Different Rating Scales" on page 8-4

External Websites

Credit Risk Modeling with MATLAB (53 min 09 sec)
Forecasting Corporate Default Rates with MATLAB (54 min 36 sec)

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 = 8×8

   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
```

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 = 8×8
```

```
   93.1172    5.8427    0.8231    0.1763    0.0377    0.0012    0.0001    0.0017
    1.6213   93.1501    4.3584    0.6614    0.1631    0.0055    0.0004    0.0397
    0.1239    2.9027   92.2297    4.0628    0.5367    0.0661    0.0028    0.0753
    0.0236    0.2313    5.0070   90.1825    3.7986    0.4734    0.0642    0.2193
    0.0216    0.1134    0.6357    5.7959   88.9866    3.4497    0.2920    0.7050
    0.0010    0.0062    0.1081    0.8697    7.3367   86.7217    2.5171    2.4395
    0.0002    0.0011    0.0120    0.2591    1.4340    4.3034   81.3027   12.6875
         0         0         0         0         0         0         0  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 = 7×5

    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
```

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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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 the comma-separated pair consisting of `'snapsPerYear'` and 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 the comma-separated pair consisting of `'transInterval'` and 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.

More About

Cohort Estimation

The cohort algorithm estimates the transition probabilities based on a sequence of snapshots of credit ratings at regularly spaced points in time.

If the credit rating of a company changes twice between two snapshot dates, the intermediate rating is overlooked and only the initial and final ratings influence the estimates.

Duration Estimation

Unlike the cohort method, the duration algorithm estimates the transition probabilities based on the full credit ratings history, looking at the exact dates on which the credit rating migrations occur.

There is no concept of snapshots in this method, and all credit rating migrations influence the estimates, even when a company's rating changes twice within a short time.

Version History

Introduced in R2010b

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

`transprobgrouptotals` | `transprob`

Topics

"Estimation of Transition Probabilities" on page 8-2
"Estimate Transition Probabilities for Different Rating Scales" on page 8-4

External Websites

Credit Risk Modeling with MATLAB (53 min 09 sec)
Forecasting Corporate Default Rates with MATLAB (54 min 36 sec)

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 = 8×8

   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
```

Obtain the credit quality thresholds.

```
thresh = transprobtothresholds(transMat)
```

```
thresh = 8×8

       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
```

Recover the transition probabilities.

```
trans = transprobfromthresholds(thresh)
```

```
trans = 8×8

   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
```

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.

Version History

Introduced in R2011b

References

[1] Gupton, G. M., C. C. Finger, and M. Bhatia. *"CreditMetrics."* Technical Document, RiskMetrics Group, Inc., 2007.

See Also

transprobtothresholds | transprob | transprobbytotals

Topics
"Estimation of Transition Probabilities" on page 8-2
"Estimate Transition Probabilities for Different Rating Scales" on page 8-4
"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)

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 = 8×8

   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
```

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 = 3×3

   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 = 2×5

    0.1056    0.2521    0.4359    0.6537    0.9027
    3.1153    6.0157    8.7179   11.2373   13.5881
```

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'
```

More About

Cohort Estimation

The cohort algorithm estimates the transition probabilities based on a sequence of snapshots of credit ratings at regularly spaced points in time.

If the credit rating of a company changes twice between two snapshot dates, the intermediate rating is overlooked and only the initial and final ratings influence the estimates. For more information, see 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.

Version History

Introduced in R2011b

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 8-13
"Estimation of Transition Probabilities" on page 8-2
"Estimate Transition Probabilities for Different Rating Scales" on page 8-4
"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)

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 = 8×8

   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
```

Estimate transition probabilities with the `'cohort'` algorithm.

```
transMatCoh = transprob(prepData,'algorithm','cohort')

transMatCoh = 8×8

   93.1345    5.9335    0.7456    0.1553    0.0311         0         0         0
    1.7359   92.9198    4.5446    0.6046    0.1560         0         0    0.0390
    0.1268    2.9716   91.9913    4.3124    0.4711    0.0544         0    0.0725
    0.0210    0.3785    5.0683   89.7792    4.0379    0.4627    0.0421    0.2103
    0.0221    0.1105    0.6851    6.2320   88.3757    3.6464    0.2873    0.6409
         0         0    0.0761    0.7230    7.9909   86.1872    2.7397    2.2831
         0         0         0    0.3094    1.8561    4.5630   80.8971   12.3743
         0         0         0         0         0         0         0  100.0000
```

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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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 the comma-separated pair consisting of `'labels'` and 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.

Version History

Introduced in R2011b

See Also

`transprob` | `transprobbytotals` | `table`

Topics

"Group Credit Ratings" on page 8-13
"Estimation of Transition Probabilities" on page 8-2
"Estimate Transition Probabilities for Different Rating Scales" on page 8-4
"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)

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 = 8×8

   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
```

Obtain the credit quality thresholds.

```
thresh = transprobtothresholds(transMat)
```

```
thresh = 8×8

       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
```

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 ≦ 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.

Version History

Introduced in R2011b

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 8-43
"Group Credit Ratings" on page 8-13
"Estimation of Transition Probabilities" on page 8-2
"Estimate Transition Probabilities for Different Rating Scales" on page 8-4
"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)

tsaccel

Acceleration between times

Syntax

```
acceleration = tsaccel(Data)
acceleration = tsaccel( ___ ,Name,Value)
```

Description

`acceleration = tsaccel(Data)` calculates the acceleration of a data series with time distance of *n* periods.

n periods.

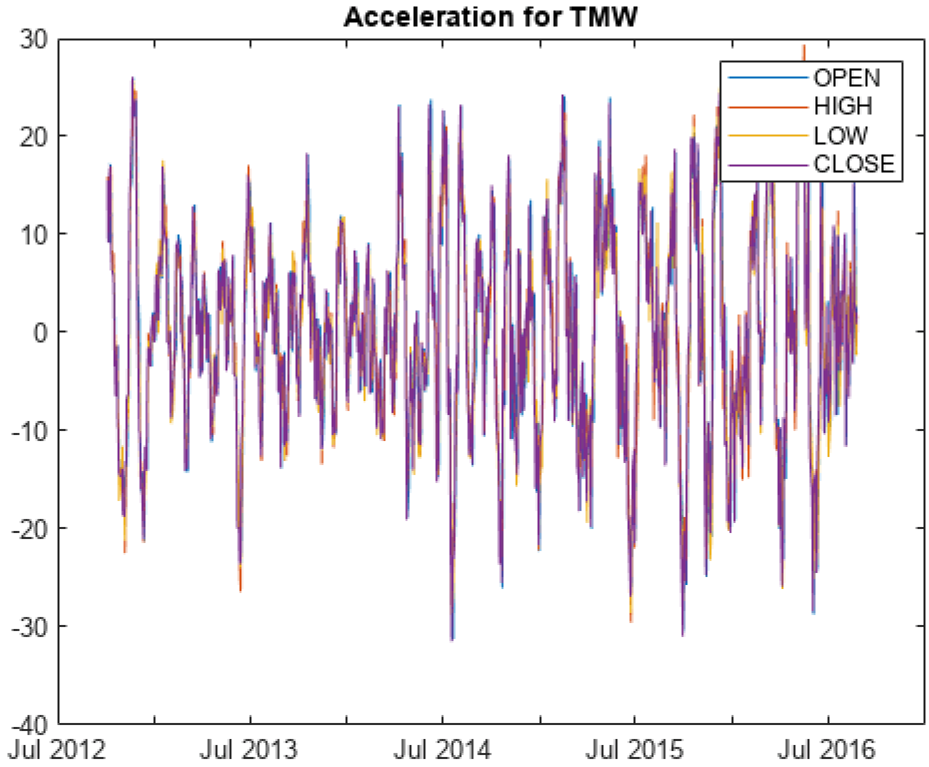
`acceleration = tsaccel( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Calculate the Acceleration of a Data Series for a Stock

Load the file `SimulatedStock.mat`, which provides a timetable (`TMW`) for financial data for TMW stock.

```
load SimulatedStock.mat
TMW.Volume = []; % remove VOLUME field
acceleration = tsaccel(TMW);
plot(acceleration.Time,acceleration.Variables)
legend('OPEN','HIGH','LOW','CLOSE')
title('Acceleration for TMW')
```



Input Arguments

`Data` — Data with high, low, open, close information

vector | matrix | table | timetable

Data with high, low, open, close information, specified as a vector, matrix, table, or timetable. For vector input, `Data` is a column vector of high, low, open, and closing prices stored in the corresponding columns. For matrix input, `Data` is an `M`-by-`N` column-oriented matrix of high, low, open, and closing prices stored in the corresponding columns. Timetables and tables with `M` rows must contain variables named `'High'`, `'Low'`, `'Open'`, and `'Close'` (case insensitive).

Data Types: `double` | `table` | `timetable`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `accelearation = tsaccel(TMW,'NumPeriods',10,'Datatype',1)`

`NumPeriods` — Period difference for acceleration

`12` (default) | positive integer

Period difference for acceleration, specified as the comma-separated pair consisting of `'NumPeriods'` and a scalar positive integer.

Data Types: `double`

`Datatype` — Indicates if `Data` contains the data itself or the momentum of the data
`0` (`Data` contains the data itself) (default) | integer with values `0` or `1`

Indicates if `Data` contains the data itself or the momentum of the data, specified as the comma-separated pair consisting of `'Datatype'` and a scalar integer with a value of:

- `0` – `Data` contains the data itself.
- `1` – `Data` contains the momentum of the data.

Data Types: `double`

Output Arguments

`acceleration` — Acceleration series
vector | matrix | table | timetable

Acceleration series, returned with the same number of rows (`M`) and columns (`N`) and the same type (vector, matrix, table, or timetable) as the input `Data`.

More About

Acceleration

Acceleration is defined as the difference of two momentum series separated by *n* periods.

Acceleration is the difference of the current momentum with the momentum *n* periods ago. By default, acceleration is based on 12-period difference.

Version History

Introduced before R2006a

R2023a: `fints` support removed for `Data` input argument
Behavior changed in R2023a

`fints` object support for the `Data` input argument is removed.

R2022b: Support for negative price data
Behavior changed in R2022b

The `Data` input accepts negative prices.

References

[1] Kaufman, P. J. *The New Commodity Trading Systems and Methods.* John Wiley and Sons, New York, 1987.

See Also

`timetable` | `table`

Topics

“Use Timetables in Finance” on page 11-7
“Convert Financial Time Series Objects (fints) to Timetables” on page 11-2

tsmom

Momentum between times

Syntax

```
momentum = tsmom(Data)
momentum = tsmom(___,Name,Value)
```

Description

`momentum = tsmom(Data)` calculates the momentum of a data series with time distance of *n* periods.

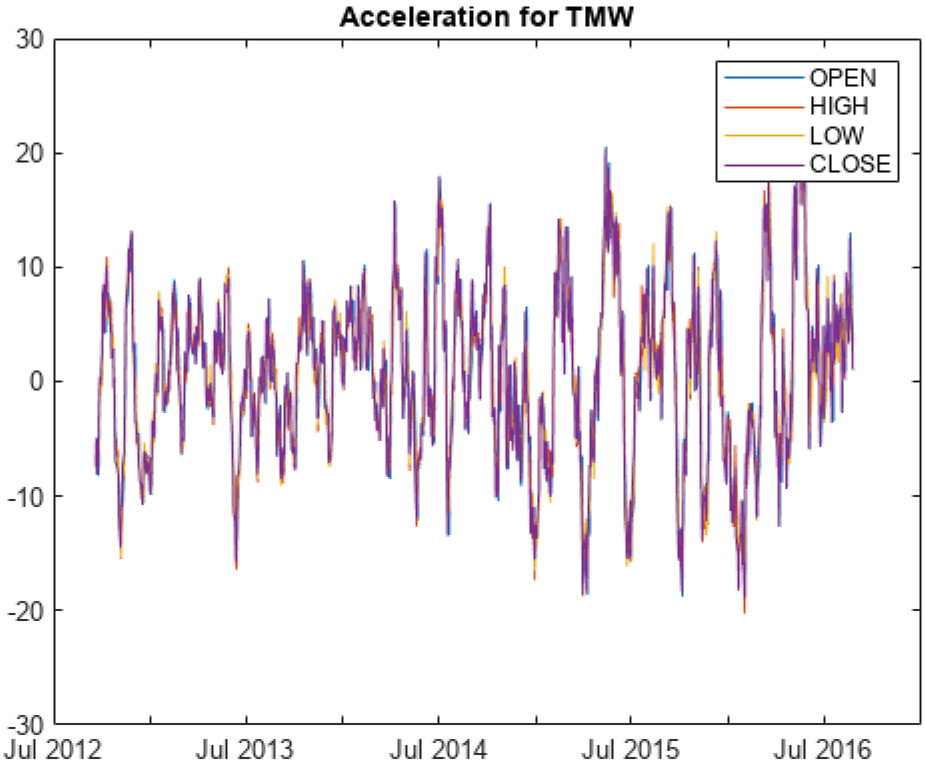
`momentum = tsmom(___,Name,Value)` adds optional name-value pair arguments.

Examples

Calculate the Momentum for a Data Series for a Stock

Load the file `SimulatedStock.mat`, which provides a timetable (`TMW`) for financial data for TMW stock.

```
load SimulatedStock.mat
TMW.Volume = []; % remove VOLUME field
momentum = tsmom(TMW);
plot(momentum.Time,momentum.Variables)
legend('OPEN','HIGH','LOW','CLOSE')
title('Acceleration for TMW')
```



Input Arguments

`Data` — Data with high, low, open, close information

vector | matrix | table | timetable

Data with high, low, open, close information, specified as a vector, matrix, table, or timetable. For vector input, `Data` is a column vector. For matrix input, `Data` is an `M`-by-`N` column oriented matrix. Timetables and tables with `M` rows can contain variables named `'High'`, `'Low'`, `'Open'`, and `'Close'` (case insensitive).

Data Types: `double` | `table` | `timetable`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `momentum = tsmom(TMW,'NumPeriods',15)`

`NumPeriods` — Period difference for momentum

`12` (default) | positive integer

Period difference for momentum, specified as the comma-separated pair consisting of `'NumPeriods'` and a scalar positive integer.

Data Types: `double`

Output Arguments

`momentum` — Momentum series
matrix | table | timetable

Momentum series, returned with the same number of rows (`M`) and the same type (matrix, table, or timetable) as the input `Data`.

More About

Momentum Series

Momentum series is the difference of the current data with the data *n* periods ago. By default, momentum is based on 12-period difference.

Version History

Introduced before R2006a

R2023a: `fints` support removed for `Data` input argument

Behavior changed in R2023a

`fints` object support for the `Data` input argument is removed.

R2022b: Support for negative price data

Behavior changed in R2022b

The `Data` input accepts negative prices.

References

[1] Kaufman, P. J. *The New Commodity Trading Systems and Methods*. John Wiley and Sons, New York, 1987.

See Also

`timetable` | `table` | `tsaccel`

Topics

“Use Timetables in Finance” on page 11-7
“Convert Financial Time Series Objects (fints) to Timetables” on page 11-2

typprice

Typical price

Syntax

```
TypicalPrice = typprice(Data)
```

Description

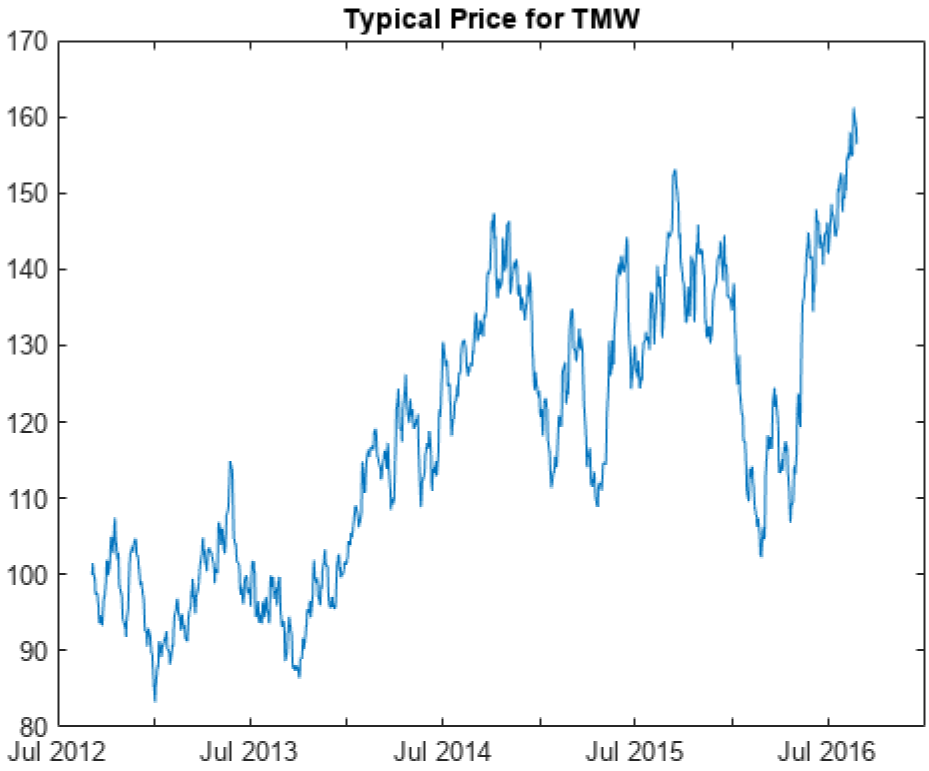
`TypicalPrice = typprice(Data)` calculates the typical prices from the series of high, low, and closing prices. The typical price is the average of the high, low, and closing prices for each period.

Examples

Calculate the Typical Price for a Data Series for a Stock

Load the file `SimulatedStock.mat`, which provides a timetable (`TMW`) for financial data for TMW stock.

```
load SimulatedStock.mat
TypicalPrice = typprice(TMW);
plot(TypicalPrice.Time,TypicalPrice.TypicalPrice)
title('Typical Price for TMW')
```



Input Arguments

`Data` — Data for high, low, and closing prices

matrix | table | timetable

Data for high, low, and closing prices, specified as a matrix, table, or timetable. For matrix input, `Data` is an `M`-by-3 matrix of high, low, and closing prices stored in the corresponding columns. Timetables and tables with `M` rows must contain variables named `'High'`, `'Low'`, and `'Close'` (case insensitive).

Data Types: `double` | `table` | `timetable`

Output Arguments

`TypicalPrice` — Typical price series

matrix | table | timetable

Typical price series, returned with the same number of rows (`M`) and the same type (matrix, table, or timetable) as the input `Data`.

Version History

Introduced before R2006a

R2023a: `fints` support removed for `Data` input argument

Behavior changed in R2023a

`fints` object support for the `Data` input argument is removed.

R2022b: Support for negative price data

Behavior changed in R2022b

The `Data` input accepts negative prices.

References

[1] Achelis, S. B. *Technical Analysis from A to Z.* Second Edition. McGraw-Hill, 1995, pp. 291–292.

See Also

`timetable` | `table` | `medprice` | `wclose`

Topics

"Use Timetables in Finance" on page 11-7
"Convert Financial Time Series Objects (fints) to Timetables" on page 11-2

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.

Note As an alternative to `uicalendar`, you can use `uidatepicker`.

Examples

Use `uicalendar` with an `uicontrol`

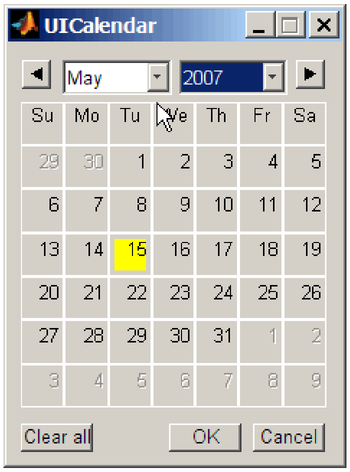
Create an `uicontrol`:

```
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'})
```

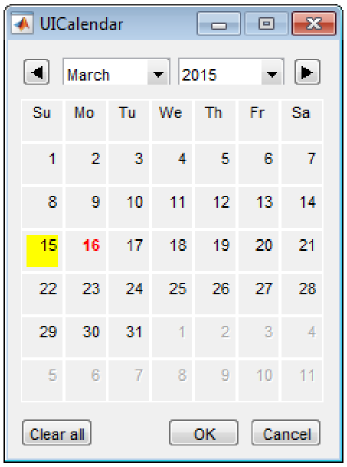


Su	Mo	Tu	We	Th	Fr	Sa
29	30	1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31	1	2
3	4	5	6	7	8	9

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'))
```



Su	Mo	Tu	We	Th	Fr	Sa
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31	1	2	3	4
5	6	7	8	9	10	11

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 12-4.

Input Arguments

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

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
datetime array | string array | date character vector | serial date number

Holiday dates in calendar, specified using a scalar or vector using a datetime array, string array, date character vectors, or serial date numbers. The corresponding date of the holiday appears Red.

Data Types: `double` | `char` | `string` | `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 `Modal`

Window figure properties, specified with using a character vector with a value of `Normal` or `Modal`. 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`

Version History

Introduced before R2006a

See Also

`holidays` | `datetime` | `uidatepicker`

Topics

"Handle and Convert Dates" on page 2-2
"Trading Calendars User Interface" on page 12-2
"UICalendar User Interface" on page 12-4

volarea

Price and volume chart

Note volarea is updated to accept data input as a matrix, timetable, or table.

The syntax for volarea has changed. Previously, when using table input, the first column of dates could be a datetime array, date character vectors, or serial date numbers, and you were required to have specific number of columns.

When using table input, the new syntax for volarea supports:

- No need for time information. If you want to pass in date information, use timetable input.
- No requirement of specific number of columns. However, you must provide valid column names. volarea must contain a column named 'price' (case insensitive).

Syntax

```
volarea(Data)
h = volarea(ax,Data)
```

Description

volarea(Data) plots a chart from a series of price and traded volume of a security.

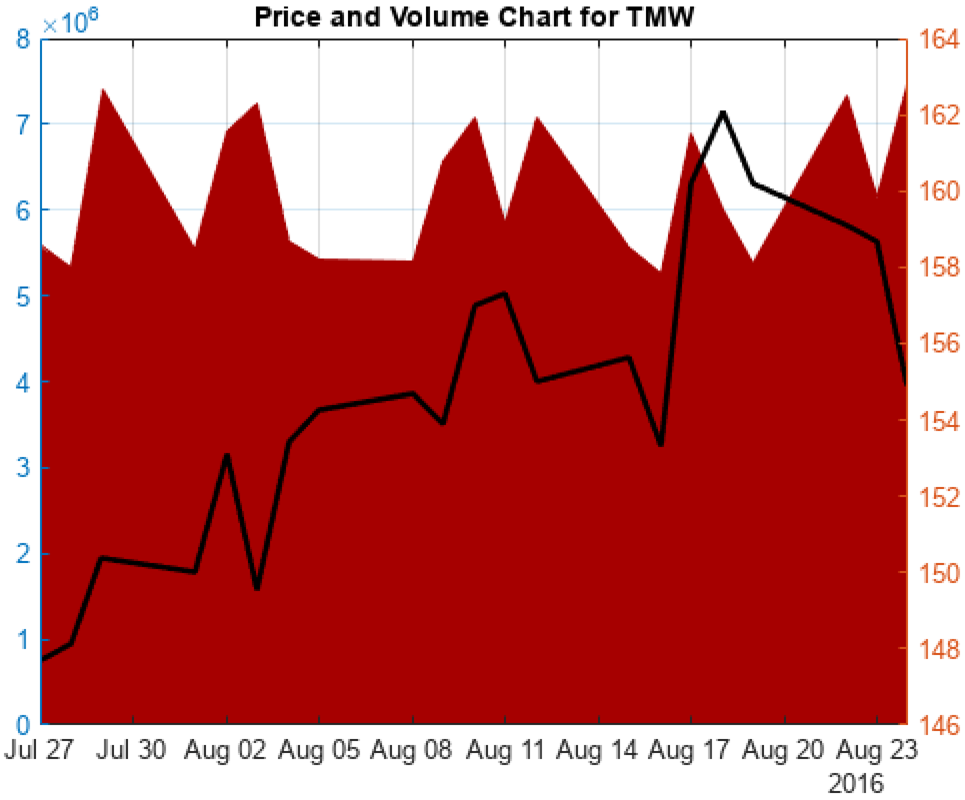
h = volarea(ax,Data) adds an optional argument for ax.

Examples

Plot Asset Date, Price, and Volume on a Single Axis for a Stock

Load the file SimulatedStock.mat, which provides a timetable (TMW) for financial data for TMW stock. This example shows how to plot asset date, price, and volume on a single axis, given asset TMW containing asset price and volume in most recent 21 days. Note that the variable name of asset price is be renamed to 'Price' (case insensitive).

```
load SimulatedStock.mat;
TMW.Properties.VariableNames{'Close'} = 'Price';
volarea(TMW(end-20:end,:))
title('Price and Volume Chart for TMW')
```



Input Arguments

`Data` — Data for prices and traded volume

matrix | table | timetable

Data for prices and traded volume, specified as a matrix, table, or timetable. For matrix input, `Data` is an `M`-by-2 matrix of prices and traded volume. Timetables and tables with `M` rows must contain variables named `'Price'` and `'Volume'` (case insensitive).

Data Types: `double` | `table` | `timetable`

`ax` — Valid axis object

current axes (`ax = gca`) (default) | axes object

(Optional) Valid axis object, specified as an axes object. The volarea plot is created in the axes specified by `ax` instead of in the current axes (`ax = gca`). The option `ax` can precede any of the input argument combinations.

Data Types: `object`

Output Arguments

`h` — Graphic handle of the figure

handle object

Graphic handle of the figure, returned as a handle object.

Version History

Introduced in R2008a

R2022b: Support for negative price data
Behavior changed in R2022b

The `Data` input accepts negative prices.

See Also

`timetable` | `table` | `movavg` | `linebreak` | `highlow` | `kagi` | `candle` | `pointfig`

Topics
“Use Timetables in Finance” on page 11-7
“Convert Financial Time Series Objects (fints) to Timetables” on page 11-2

volroc

Volume rate of change

Syntax

```
volumeChangeRate = volroc(Data)
volumeChangeRate = volroc( ___ ,Name,Value)
```

Description

`volumeChangeRate = volroc(Data)` calculates the volume rate-of-change from a data series of volume traded. The volume rate-of-change is calculated between the current volume and the volume *n* periods ago. By default, the Volume rate of change is based on a 12-period difference.

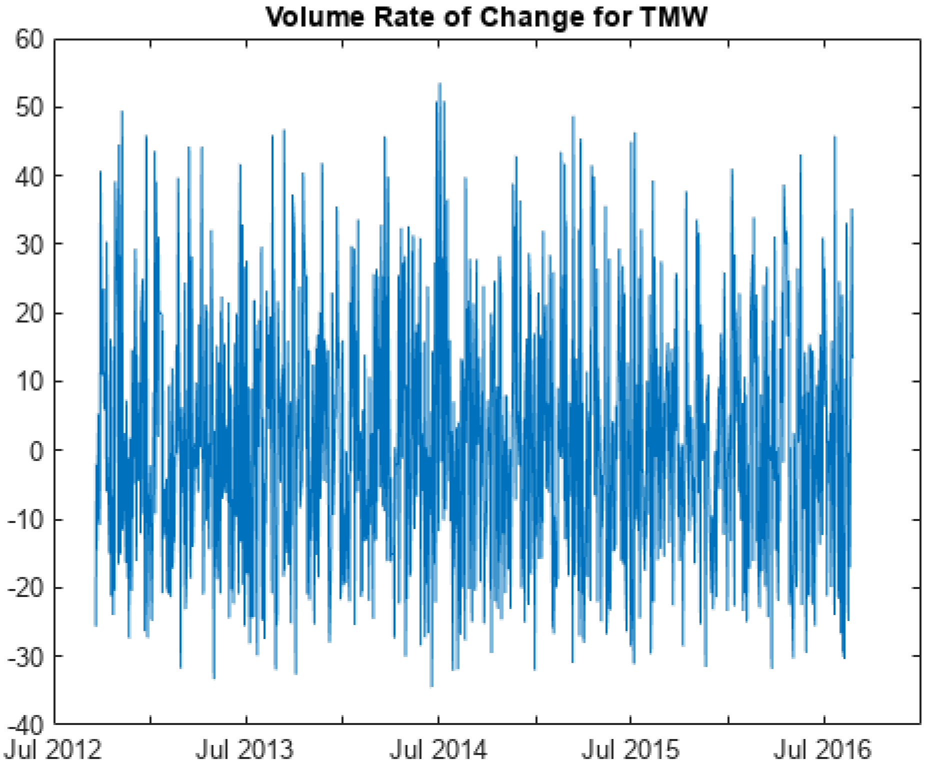
`volumeChangeRate = volroc( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Calculate the Volume Rate of Change for a Data Series for a Stock

Load the file `SimulatedStock.mat`, which provides a timetable (`TMW`) for financial data for TMW stock.

```
load SimulatedStock.mat
volumeChangeRate = volroc(TMW);
plot(volumeChangeRate.Time,volumeChangeRate.VolumeChangeRate)
title('Volume Rate of Change for TMW')
```



Input Arguments

`Data` — Data for volume traded

vector | table | timetable

Data for volume traded, specified as a vector, table, or timetable. For vector input, `Data` is an `M`-by-`1` column vector of volume traded. Timetables and tables with `M` rows must contain a variable named `'Volume'` (case insensitive).

Data Types: `double` | `table` | `timetable`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `volumeChangeRate = volroc(TMW,'NumPeriods',18)`

`NumPeriods` — Period difference for `volumeChangeRate`

`12` (default) | positive integer

Period difference for `volumeChangeRate`, specified as the comma-separated pair consisting of `'NumPeriods'` and a scalar positive integer.

Data Types: `double`

Output Arguments

`volumeChangeRate` — Volume rate of change series
vector | table | timetable

Volume rate of change series, returned with the same number of rows (`M`) and the same type (vector, table, or timetable) as the input `Data`.

Version History

Introduced before R2006a

R2023a: `fints` support removed for `Data` input argument
Behavior changed in R2023a

`fints` object support for the `Data` input argument is removed.

R2022b: Support for negative price data
Behavior changed in R2022b

The `Data` input accepts negative prices.

References

[1] Achelis, S. B. *Technical Analysis from A to Z.* Second Edition. McGraw-Hill, 1995, pp. 310–311.

See Also

`timetable` | `table` | `prcroc`

Topics
“Use Timetables in Finance” on page 11-7
“Convert Financial Time Series Objects (fints) to Timetables” on page 11-2

wclose

Weighted close

Syntax

```
WeightedClose = wclose(Data)
```

Description

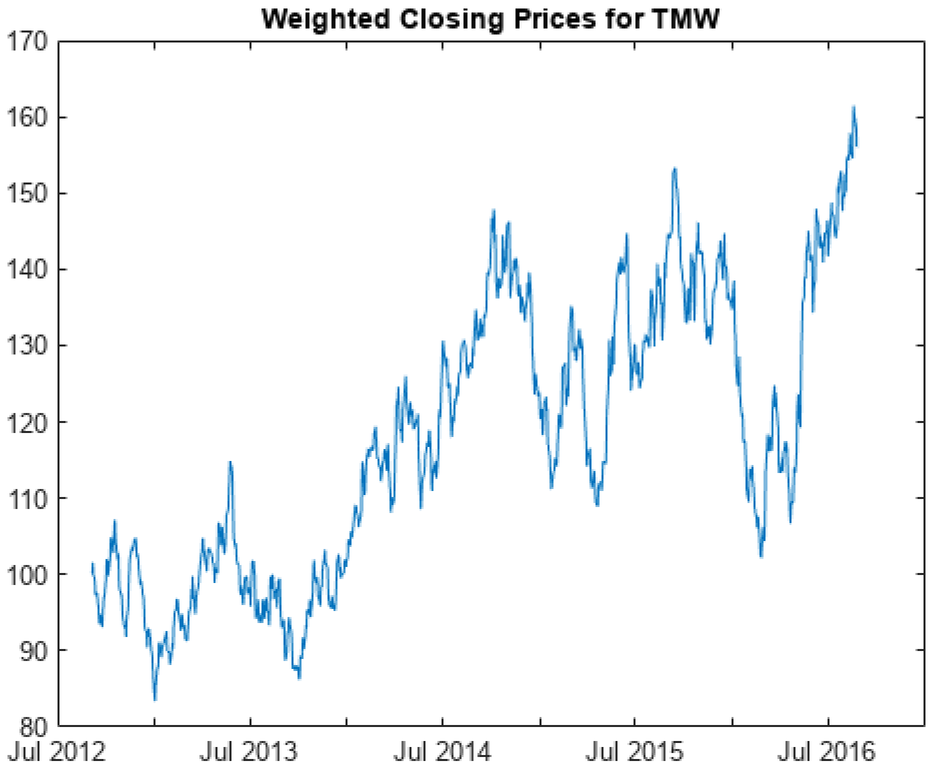
`WeightedClose = wclose(Data)` calculates the weighted closing prices from the series of high, low, and closing prices. The weighted closing price is the average of twice the closing price plus the high and low prices.

Examples

Calculate the Weighted Closing Prices for a Data Series for a Stock

Load the file `SimulatedStock.mat`, which provides a timetable (`TMW`) for financial data for TMW stock.

```
load SimulatedStock.mat
WeightedClose = wclose(TMW);
plot(WeightedClose.Time,WeightedClose.WeightedClose)
title('Weighted Closing Prices for TMW')
```



Input Arguments

`Data` — Data for high, low, and closing prices

matrix | table | timetable

Data for high, low, and closing prices, specified as a matrix, table, or timetable. For matrix input, `Data` is an `M`-by-`3` matrix of high, low, and closing prices stored in the corresponding columns. Timetables and tables with `M` rows must contain variables named `'High'`, `'Low'`, and `'Close'` (case insensitive).

Data Types: `double` | `table` | `timetable`

Output Arguments

`WeightedClose` — Weighted closing price series

matrix | table | timetable

Weighted closing price series, returned with the same number of rows (`M`) and the same type (matrix, table, or timetable) as the input `Data`.

Version History

Introduced before R2006a

R2023a: `fints` support removed for `Data` input argument

Behavior changed in R2023a

`fints` object support for the `Data` input argument is removed.

R2022b: Support for negative price data

Behavior changed in R2022b

The `Data` input accepts negative prices.

References

[1] Achelis, S. B. *Technical Analysis from A to Z.* Second Edition. McGraw-Hill, 1995, pp. 312–313.

See Also

`timetable` | `table` | `medprice` | `typprice`

Topics

"Use Timetables in Finance" on page 11-7
"Convert Financial Time Series Objects (fints) to Timetables" on page 11-2

weights2holdings

Portfolio values and weights into holdings

Syntax

```
Holdings = weights2holdings(Values,Weights,Prices)
```

Description

`Holdings = weights2holdings(Values,Weights,Prices)` converts portfolio values and weights into portfolio holdings.

Note `weights2holdings` does not create round-lot positions. The output `Holdings` are floating-point values.

Input Arguments

`Values` — Portfolio values

scalar numeric | vector

Portfolio values, specified as a scalar or an `NPORTS` vector.

Data Types: `double`

`Weights` — Portfolio weights

matrix

Portfolio weights, specified as an `NPORTS`-by-`NASSETS` matrix. The weights sum to the value of a `Budget` constraint, which is usually 1. (See `holdings2weights` for information about budget constraints.)

Data Types: `double`

`Prices` — Asset prices

vector

Asset prices, specified as an `NASSETS` vector.

Data Types: `double`

Output Arguments

`Holdings` — Holdings value

matrix

Holdings value, returned as an `NPORTS`-by-`NASSETS` matrix that contains the holdings of `NPORTS` portfolios that contain `NASSETS` assets.

Version History

Introduced before R2006a

See Also

`holdings2weights`

willad

Williams Accumulation/Distribution line

Syntax

```
WADLine = willad(Data)
```

Description

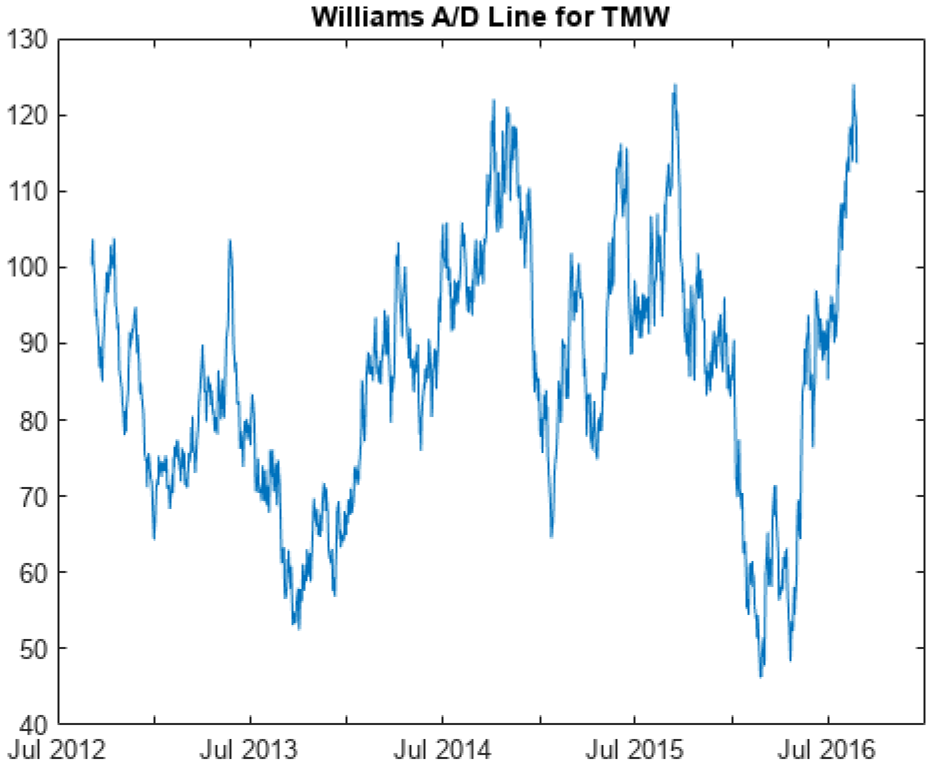
`WADLine = willad(Data)` calculates the Williams Accumulation/Distribution line from the series of high, low, and closing prices.

Examples

Calculate the Williams Accumulation/Distribution Line for a Data Series for a Stock

Load the file `SimulatedStock.mat`, which provides a timetable (`TMW`) for financial data for TMW stock.

```
load SimulatedStock.mat
WADLine = willad(TMW);
plot(WADLine.Time,WADLine.WillAD)
title('Williams A/D Line for TMW')
```



Input Arguments

`Data` — Data for high, low, and closing prices

matrix | table | timetable

Data for high, low, and closing prices, specified as a matrix, table, or timetable. For matrix input, `Data` is an `M`-by-`3` matrix of high, low, and closing prices stored in the corresponding columns. Timetables and tables with `M` rows must contain variables named `'High'`, `'Low'`, and `'Close'` (case insensitive).

Data Types: `double` | `table` | `timetable`

Output Arguments

`WADLine` — Williams Accumulation/Distribution line

matrix | table | timetable

Williams Accumulation/Distribution line, returned with the same number of rows (`M`) and the same type (matrix, table, or timetable) as the input `Data`.

Version History

Introduced before R2006a

R2023a: `fints` support removed for `Data` input argument

Behavior changed in R2023a

`fints` object support for the `Data` input argument is removed.

R2022b: Support for negative price data

Behavior changed in R2022b

The `Data` input accepts negative prices.

References

[1] Achelis, S. B. *Technical Analysis from A to Z.* Second Edition. McGraw-Hill, 1995, pp. 314–315.

See Also

`timetable` | `table` | `adline` | `adosc` | `willpctr`

Topics

“Use Timetables in Finance” on page 11-7
“Convert Financial Time Series Objects (fints) to Timetables” on page 11-2

willpctr

Williams %R

Syntax

```
PercentR = willpctr(Data)
PercentR = willpctr( ___ ,Name,Value)
```

Description

`PercentR = willpctr(Data)` calculates the Williams PercentR (%R) values for a data series of with high, low, and closing prices.

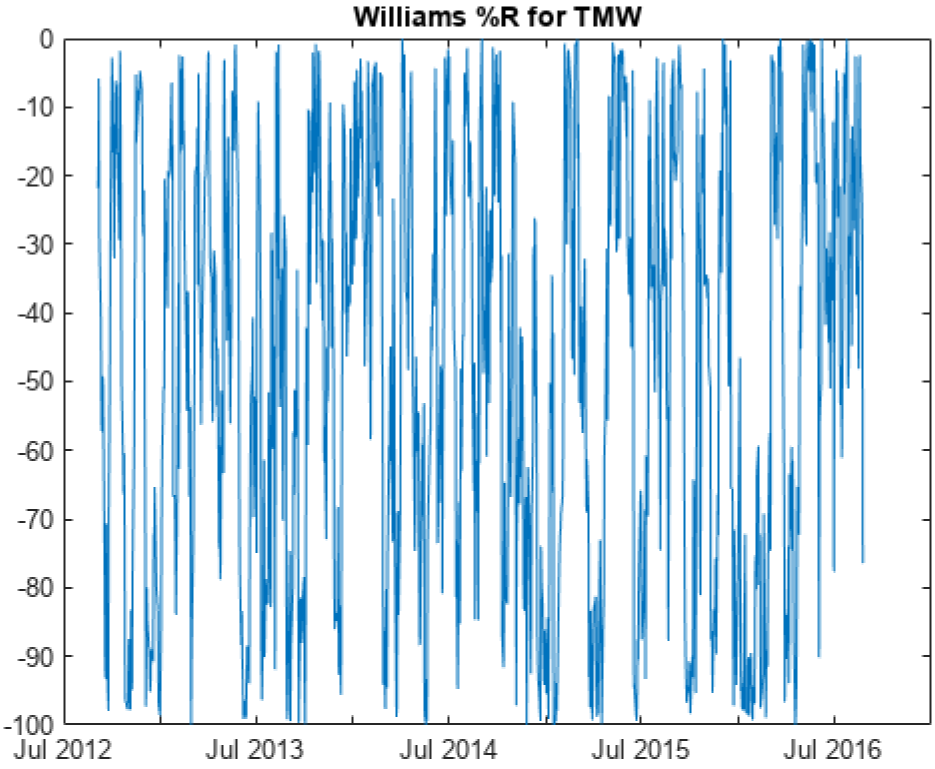
`PercentR = willpctr( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Calculate the Williams %R for a Data Series for a Stock

Load the file `SimulatedStock.mat`, which provides a timetable (`TMW`) for financial data for TMW stock.

```
load SimulatedStock.mat
PercentR = willpctr(TMW);
plot(PercentR.Time,PercentR.WillPercentR)
title('Williams %R for TMW')
```



Input Arguments

`Data` — Data with high, low, and close information

matrix | table | timetable

Data with high, low, open, close information, specified as a matrix, table, or timetable. For matrix input, `Data` is an `M`-by-`3` with high, low, and closing prices stored in the corresponding columns. Timetables and tables with `M` rows must contain variables named `'High'`, `'Low'`, and `'Close'` (case insensitive).

Data Types: `double` | `table` | `timetable`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `PercentR = willpctr(TMW,'NumPeriods',15)`

`NumPeriods` — Moving window for Williams PercentR

`14` (default) | positive integer

Moving window for Williams PercentR, specified as the comma-separated pair consisting of `'NumPeriods'` and a scalar positive integer.

Data Types: `double`

Output Arguments

`PercentR` — Williams PercentR series
matrix | table | timetable

Williams PercentR series, returned with the same number of rows (`M`) and the same type (matrix, table, or timetable) as the input `Data`.

More About

Williams %R

Williams %R shows the current closing price in relation to the high and low of the past *n* days.

By default, Williams %R values are based on 14 periods.

Version History

Introduced before R2006a

R2023a: `fints` support removed for `Data` input argument
Behavior changed in R2023a

`fints` object support for the `Data` input argument is removed.

R2022b: Support for negative price data
Behavior changed in R2022b

The `Data` input accepts negative prices.

References

[1] Achelis, S. B. *Technical Analysis from A to Z.* Second Edition. McGraw-Hill, 1995, pp. 316–317.

See Also

`timetable` | `table` | `willad` | `stochosc`

Topics

"Use Timetables in Finance" on page 11-7
"Convert Financial Time Series Objects (fints) to Timetables" on page 11-2

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 a datetimes for array for `StartDate` and `EndDate`.

```
Days = wrkdydif(datetime(2000,9,1), datetime(2000,9,11), 1)
```

```
Days = 6
```

Input Arguments

`StartDate` — Start date

datetime array | string array | date character vector

Start date, specified as an `N`-by-`1` or `1`-by-`N` vector using a datetime array, string array, or date character vectors.

To support existing code, `wrkdydif` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`EndDate` — End date

datetime array | string array | date character vector

End date, specified as an `N`-by-`1` or `1`-by-`N` vector using a datetime array, string array, or date character vectors.

To support existing code, `wrkdydif` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | datetime

Holidays — Number of holidays between StartDate and EndDate
vector of integers

Number of holidays between the StartDate and EndDate, specified as an N-by-1 or 1-by-N vector of integers.

Data Types: 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.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended
Not recommended starting in R2022b

Although wrkdydif supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

busdate | datewrkdy | days365 | daysact | daysdif | holidays | yearfrac | datetime

Topics
“Handle and Convert Dates” on page 2-2
“Trading Calendars User Interface” on page 12-2
“UICalendar User Interface” on page 12-4

xirr

Internal rate of return for nonperiodic cash flow

Syntax

```
Return = xirr(CashFlow,CashFlowDates)
Return = xirr( ___ ,Guess,MaxIterations,Basis)
```

Description

`Return = xirr(CashFlow,CashFlowDates)` returns the internal rate of return for a schedule of nonperiodic cash flows.

`Return = xirr( ___ ,Guess,MaxIterations,Basis)` adds optional arguments.

Examples

Find Internal Rate of Return for Nonperiodic Cash Flow

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 12-Jan-2007

2500 14-Feb-2008

2000 03-Mar-2008

3000 14-Jun-2008

4000 01-Dec-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)
```

```
Return = 0.1006
```

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)
```

```
Return = 0.1006
```

Input Arguments

CashFlow — Cash flow

vector | matrix

Cash flow, specified as a vector or matrix. The first entry is the initial investment. 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, usually entered as a negative number.

Data Types: double

CashFlowDates — Cash flow dates

datetime array | string array | date character vector

Cash flow dates, specified as a vector or matrix using a datetime array, string array, or date character vectors. 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.

To support existing code, xirr also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | datetime | string

Guess — Initial estimate of the internal rate of return

0.1 (10%) (default) | numeric

(Optional) Initial estimate of the internal rate of return, specified as a scalar or vector. If Guess is a scalar, then it is applied to all streams, and if Guess is a vector, then it is the same length as the number of streams.

Data Types: double

MaxIterations — Number of iterations used by Newton's method to solve the internal rate of return

50 (default) | positive integer

(Optional) Number of iterations used by Newton's method to solve the internal rate of return, specified as a scalar or vector of positive integers. If MaxIterations is a scalar, then it is applied to all streams, and if MaxIterations is a vector, then it is the same length as the number of streams.

Data Types: double

Basis — Day-count basis

0 (actual/actual) (default) | positive integers of the set [1...13] | vector of positive integers of the set [1...13]

(Optional) Day-count basis, specified as a positive integer using scalar or a N-by-1 vector. If Basis is a scalar, then it is applied to all streams, and if Basis is a vector, then it is the same length as the number of streams.

- 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 2-16.

Data Types: double

Output Arguments

Return — Annualized internal rate of return of each cash flow stream

numeric

Annualized internal rate of return of each cash flow stream, returned as a vector. A NaN indicates that a solution was not found.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although xirr supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Brealey and Myers. *Principles of Corporate Finance.* McGraw-Hill Higher Education, Chapter 5, 2003.

[2] Sharpe, William F., and Gordon J. Alexander. *Investments.* Englewood Cliffs, NJ: Prentice-Hall. 4th ed., 1990.

See Also

`fvvar` | `irr` | `mirr` | `pvvar` | `datetime`

Topics

"Analyzing and Computing Cash Flows" on page 2-11

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
```

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 2-16.

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.

Version History

Introduced before R2006a

See Also

`days360` | `days365` | `daysact` | `yearfrac`

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

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 `M`-by-`NUMINST` conforming matrices 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 the Date Range Spans a Leap Year

Use a `Basis` of 12 to compute `yearfrac` when parts of the date range are in leap year and in a non-leap year. The output `YearFraction` is a fraction, in years, based on the number of days between the `StartDate` and `EndDate`.

```
YearFraction = yearfrac('1-Jan-2016','30-Jan-2017',12)
```

```
YearFraction = 1.0795
```

Compute `yearfrac` When Specifying datetimes

Given a `Basis` of `9`, compute `yearfrac` when the `StartDate` and `EndDate` are specified using datetimes.

```
yearfrac(datetime(2000,1,1), datetime(2001,1,1), 9)
```

```
ans = 1.0167
```

Input Arguments

`StartDate` — Start date
datetime array | string array | date character vector

Start date, specified as a scalar or an `M`-by-`N` vector using a datetime array, string array, or date character vectors.

Note The dimensions of the `StartDate`, `EndDate`, and `Basis` vectors must be consistent.

To support existing code, `yearfrac` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`EndDate` — End date
datetime array | string array | date character vector

End date, specified as a scalar or an `M`-by-`N` vector using a datetime array, string array, or date character vectors.

Note The dimensions of the `EndDate`, `StartDate`, and `Basis` vectors must be consistent.

To support existing code, `yearfrac` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `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 a scalar or an `M`-by-`N` matrix of integers with values of `0` through `13`

Note The dimensions of the `Basis`, `StartDate`, and `EndDate` matrices must be consistent.

.

- 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 2-16.

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 as a scalar or an M-by-N matrix.

More About

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
```

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although yearfrac supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)

y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`days360` | `date2time` | `days365` | `daysact` | `daysdif` | `wrkdydif` | `yeardays` | `datetime`

Topics

“Handle and Convert Dates” on page 2-2
“Trading Calendars User Interface” on page 12-2
“UICalendar User Interface” on page 12-4

ylddisc

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 = datetime(2000,10,14);
Maturity = datetime(2001,3,17);
Face = 100;
Price = 96.28;
Basis = 2;

Yield = ylddisc(Settle, Maturity, Face, Price, Basis)
```

```
Yield = 0.0903
```

Input Arguments

`Settle` — Settlement date of security

datetime array | string array | date character vector

Settlement date of the security, specified using a datetime array, string array, or date character vectors. The `Settle` date must be before the `Maturity` date.

To support existing code, `ylddisc` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Maturity` — Maturity date of security

datetime array | string array | date character vector

Maturity date of the security, specified using a datetime array, string array, or date character vectors.

To support existing code, `ylddisc` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Face` — Redemption value of security

numeric

Redemption value (par value) of the security, specified as a scalar or vector using numeric values.

Data Types: `double`

`Price` — Discount price of security

numeric

Discount price of the security, specified as a scalar or vector using numeric values.

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 as a scalar or vector 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 2-16.

Data Types: `double`

Output Arguments

`Yield` — Yield of discounted security
numeric

Yield of discounted security, returned as a scalar or vector of numeric values.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended
Not recommended starting in R2022b

Although `ylddisc` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

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

See Also

`acrudisc` | `bndprice` | `bndyield` | `prdisc` | `yldmat` | `yldtbill` | `datetime`

Topics
“Yield Functions” on page 2-22
“Yield Conventions” on page 2-21

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 = datetime(2000,2,7);
Maturity = datetime(2000,4,13);
Issue = datetime(1999,10,11);
Face = 100;
Price = 99.98;
CouponRate = 0.0608;
Basis = 1;

Yield = yldmat(Settle, Maturity, Issue, Face, Price, CouponRate, Basis)
Yield = 0.0607
```

Input Arguments

`Settle` — Settlement date of security

datetime array | string array | date character vector

Settlement date of the security, specified using a datetime array, string array, or date character vectors. The `Settle` date must be before the `Maturity` date.

To support existing code, `yldmat` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Maturity` — Maturity date of security

datetime array | string array | date character vector

Maturity date of the security, specified using a datetime array, string array, or date character vectors.

To support existing code, `yldmat` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Issue` — Issue date of security

datetime array | string array | date character vector

Issue date of the security, specified using a datetime array, string array, or date character vectors.

To support existing code, `yldmat` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Face` — Redemption value of security

numeric

Redemption value (par value) of the security, specified as a scalar or vector using numeric values.

Data Types: `double`

`Price` — Price of security

numeric

Price of the security, specified as a scalar or vector using numeric values.

Data Types: `double`

`CouponRate` — Coupon rate of security

decimal fraction

Coupon rate of the security, specified as a scalar or a vector using decimal fractions.

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 vector 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 2-16.

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 scalar or vector of numeric values.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although `yldmat` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

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

See Also

`acrudisc` | `bndprice` | `bndyield` | `prmat` | `ylddisc` | `yldtbill` | `datetime`

Topics

"Yield Functions" on page 2-22
"Yield Conventions" on page 2-21

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(2000,2,10), datetime(2000,8,6), 1000, 981.36)
```

```
Yield = 0.0384
```

Input Arguments

`Settle` — Settlement date of Treasury bill

datetime array | string array | date character vector

Settlement date of the Treasury bill, specified using a datetime array, string array, or date character vectors. The `Settle` date must be before the `Maturity` date.

To support existing code, `yldtbill` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Maturity` — Maturity date of Treasury bill
datetime array | string array | date character vector

Maturity date of the Treasury bill, specified using a datetime array, string array, or date character vectors.

To support existing code, `yldtbill` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `char` | `string` | `datetime`

`Face` — Redemption value of Treasury bill
numeric

Redemption value (par value) of the Treasury bill, specified as a scalar or vector using numeric values.

Data Types: `double`

`Price` — Price of Treasury bill
numeric

Price of the Treasury bill, specified as a scalar or vector using numeric values.

Data Types: `double`

Output Arguments

`Yield` — Yield for Treasury bill
numeric

Yield for Treasury bill, returned as a scalar or vector of numeric values.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although `yldtbill` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

References

[1] Bodie, Kane, and Marcus. *Investments.* McGraw-Hill Education, 2013.

See Also

`beytbill` | `bndyield` | `prtbill` | `yldmat` | `datetime`

Topics

"Computing Treasury Bill Price and Yield" on page 2-26
"Yield Functions" on page 2-22
"Treasury Bills Defined" on page 2-25
"Yield Conventions" on page 2-21

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 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 = datetime(1997,12,18);
OutputCompounding = 2;

t=array2table(Bonds);
t.Bonds1 = datetime(t.Bonds1,'ConvertFrom','datenum','Locale','en_US');
[ZeroRates, CurveDates] = zbtprice(t, Prices, Settle, OutputCompounding)
```

```
ZeroRates = 11×1

    0.0616
    0.0609
    0.0658
    0.0590
    0.0647
    0.0655
    0.0606
    0.0601
    0.0642
    0.0621
      ⋮


CurveDates = 11x1 datetime
   01-Jun-1998
   01-Sep-1999
   15-Nov-1999
   15-Jan-2000
   30-Jun-2000
   01-Jul-2000
   30-Apr-2001
   30-Jun-2001
   01-Jul-2001
   15-Apr-2002
   30-Apr-2002
```

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 a datetime array, string array, or date character vectors.
- `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 2-16.
- `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 a datetime array, string array, or date character vectors.
- 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` | `char` | `string` | `datetime` | `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

datetime scalar | string scalar | date character vector

Settlement date representing time zero in derivation of zero curve, specified as scalar datetime, string, or date character vector. Settle represents time zero for deriving the zero curve, and it is normally the common settlement date for all the bonds.

To support existing code, zbtprice also accepts serial date numbers as inputs, but they are not recommended.

Data Types: char | string | 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

vector of 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

datetime | serial date number

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 Maturitydate 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. Use the function datestr to convert serial date numbers to formatted date character vectors.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although `zbtprice` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

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

`zbtyield` | `datetime`

Topics

“Term Structure of Interest Rates” on page 2-29
“Fixed-Income Terminology” on page 2-15

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 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 = datetime(1997,12,18);
OutputCompounding = 2;
t = array2table(Bonds,'VariableNames',{'Maturity','CouponRate', 'Face' ,'Period', 'Basis', 'EndMo
disp(t)

     Maturity     CouponRate    Face    Period    Basis    EndMonthRule
    __________    __________    ____    ______    _____    ____________

    7.2991e+05      0.0475      100       2         0           0      
    7.3067e+05        0.06      100       2         0           0      
    7.3067e+05     0.09375      100       6         1           0      
    7.3103e+05     0.05125      100       1         3           1      
    7.3132e+05     0.07125      100       4         1           0      
     7.305e+05       0.065      100       2         0           0      
    7.3036e+05        0.08      100       3         3           0      
    7.3097e+05     0.05875      100       2         0           0      
    7.3044e+05     0.07125      100       2         0           0      
    7.3067e+05        0.07      100       2         3           1      
    7.3103e+05      0.0525      100       2         3           0      
    7.3134e+05        0.07      100       2         0           0      

t.Maturity = datetime(t.Maturity,'ConvertFrom','datenum','Locale','en_US');
[ZeroRates, CurveDates] = zbtyield(t, Yields, Settle, OutputCompounding)

ZeroRates = 11×1

    0.0616
    0.0603
    0.0657
    0.0590
    0.0649
    0.0650
    0.0606
    0.0611
    0.0643
    0.0614
      ⋮


CurveDates = 11x1 datetime
   01-Jun-1998
   01-Sep-1999
   15-Nov-1999
   15-Jan-2000
   30-Jun-2000
   01-Jul-2000
   30-Apr-2001
   30-Jun-2001
   01-Jul-2001
   15-Apr-2002
   30-Apr-2002
```

Compute the Real Zero Rates From the Real Yields of Inflation-Linked Bonds

Use `zbtyield` to compute the real zero rates from the real yields of inflation-linked bonds.

```
% Load the data
load usbond_02Sep2008
Settle = datetime(2008,9,2);
```

Compute the real yields and then compute the real zero rates.

```
RealYields = bndyield(TIPSPrice,TIPSCoupon,Settle,TIPSMaturity);
TIPSBonds = [TIPSMaturity TIPSCoupon];
[RealZeroRates, CurveDates] = zbtyield(TIPSBonds, RealYields, Settle)
```

```
RealZeroRates = 26×1

    0.0069
    0.0094
    0.0092
    0.0111
    0.0110
    0.0119
    0.0116
    0.0128
    0.0126
    0.0136
      ⋮


CurveDates = 26x1 datetime
   15-Jan-2010
   15-Apr-2010
   15-Jan-2011
   15-Apr-2011
   15-Jan-2012
   15-Apr-2012
   15-Jul-2012
   15-Apr-2013
   15-Jul-2013
   15-Jan-2014
   15-Jul-2014
   15-Jan-2015
   15-Jul-2015
   15-Jan-2016
   15-Jul-2016
   15-Jan-2017
   15-Jul-2017
   15-Jan-2018
   15-Jul-2018
   15-Jan-2025
   15-Jan-2026
   15-Jan-2027
   15-Jan-2028
   15-Apr-2028
   15-Apr-2029
```

```
15-Apr-2032
```

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 a datetime array, string array, or date character vectors.
- 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 2-16.
- 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 a datetime array, string array, or date character vectors.

- 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` | `char` | `string` | `datetime` | `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
datetime scalar | string scalar | date character vector

Settlement date representing time zero in derivation of zero curve, specified as a scalar datetime, string, or date character vector. `Settle` represents time zero for deriving the zero curve, and it is normally the common settlement date for all the bonds.

To support existing code, `zbtyield` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `string` | `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
datetime | serial date number

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 Maturitydate 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. Use the function datestr to convert serial date numbers to formatted date character vectors.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended
Not recommended starting in R2022b

Although zbtyield supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)

y =

        2021
```

There are no plans to remove support for serial date number inputs.

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

`zbtprice` | `datetime`

Topics

“Term Structure of Interest Rates” on page 2-29
“Fixed-Income Terminology” on page 2-15

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. Use the function `datestr` to convert serial date numbers to formatted date character vectors. 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 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 = [datetime(2000,11,6)
              datetime(2000,12,11)
              datetime(2001,1,15)
              datetime(2001,2,5)
              datetime(2001,3,4)
              datetime(2001,4,2)
              datetime(2001,4,30)
              datetime(2001,6,25)
```

```
                 datetime(2001,9,4)
                 datetime(2001,11,12)];

Settle = datetime(2000,11,3);
Compounding = 365;
Basis = 3;

[DiscRates, CurveDates] = zero2disc(ZeroRates, CurveDates,...
Settle, Compounding, Basis)
```

```
DiscRates = 10×1

    0.9996
    0.9947
    0.9896
    0.9866
    0.9826
    0.9787
    0.9745
    0.9665
    0.9552
    0.9466


CurveDates = 10x1 datetime
   06-Nov-2000
   11-Dec-2000
   15-Jan-2001
   05-Feb-2001
   04-Mar-2001
   02-Apr-2001
   30-Apr-2001
   25-Jun-2001
   04-Sep-2001
   12-Nov-2001
```

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

datetime array | string array | date character vector

Maturity dates that correspond to the input ZeroRates, specified as NUMBONDS-by-1 vector using a datetime array, string array, or date character vectors.

To support existing code, zero2disc also accepts serial date numbers as inputs, but they are not recommended.

Data Types: datetime | string | char

Settle — Common settlement date for ZeroRates

datetime scalar | string scalar | date character vector

Common settlement date for ZeroRates, specified as a scalar datetime, string, or date character vector. Settle is the settlement date for the bonds from which the zero curve was bootstrapped.

To support existing code, zero2disc also accepts serial date numbers as inputs, but they are not recommended.

Data Types: datetime | string | char

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: [DiscRates,CurveDates] = zero2disc(ZeroRates,CurveDates,Settle,'Compounding',4,'Basis',6)

Compounding — Rate at which input ZeroRateszero 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 the comma-separated pair consisting of 'Compounding' and 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 as the comma-separated pair consisting of 'Basis' and 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 2-16.

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 | 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. Use the function `datestr` to convert serial date numbers to formatted date character vectors.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although `zero2disc` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`disc2zero` | `datetime` | `fwd2zero` | `prbyzero` | `pyld2zero` | `zbtprice` | `zbtyield` | `zero2fwd` | `zero2pyld`

Topics

“Term Structure of Interest Rates” on page 2-29
“Fixed-Income Terminology” on page 2-15

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. Use the function datestr to convert serial date numbers to formatted date character vectors. 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 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 = [datetime(2000,11,6)
              datetime(2000,12,11)
              datetime(2001,1,15)
              datetime(2001,2,5)
              datetime(2001,3,4)
              datetime(2001,4,2)
              datetime(2001,4,30)
              datetime(2001,6,25)
              datetime(2001,9,4)
              datetime(2001,11,12)];


Settle = datetime(2000,11,3);
InputCompounding = 1;
InputBasis = 2;
OutputCompounding = 1;
OutputBasis = 2;

[ForwardRates, CurveDates] = zero2fwd(ZeroRates, CurveDates,...
Settle,'InputCompounding',1,'InputBasis',2,'OutputCompounding',1,'OutputBasis',2)
```

```
ForwardRates = 10×1

    0.0458
    0.0506
    0.0535
    0.0522
    0.0541
    0.0498
    0.0544
    0.0531
    0.0594
    0.0476


CurveDates = 10x1 datetime
   06-Nov-2000
   11-Dec-2000
   15-Jan-2001
   05-Feb-2001
   04-Mar-2001
   02-Apr-2001
   30-Apr-2001
   25-Jun-2001
   04-Sep-2001
   12-Nov-2001
```

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

datetime array | string array | date character vector

Maturity dates, specified as a NUMBONDS-by-1 vector using a datetime array, string array, or date character vectors that correspond to the ZeroRates.

To support existing code, zero2fwd also accepts serial date numbers as inputs, but they are not recommended.

Data Types: datetime | string | char

Settle — Common settlement date for ZeroRates

datetime scalar | string scalar | date character vector

Common settlement date for input ZeroRates, specified as scalar datetime, string, or date character vector.

To support existing code, zero2fwd also accepts serial date numbers as inputs, but they are not recommended.

Data Types: datetime | string | char

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

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 as the comma-separated pair consisting of 'InputCompounding' and 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 as the comma-separated pair consisting of 'InputBasis' and 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 2-16.

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 as the comma-separated pair consisting of 'OutputCompounding' and 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 as the comma-separated pair consisting of 'OutputBasis' and 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 2-16.

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

datetime | serial date number

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.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although `zero2fwd` supports serial date numbers, `datetime` values are recommended instead. The `datetime` data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to `datetime` values, use the `datetime` function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

`fwd2zero` | `getForwardRates` | `datetime` | `disc2zero` | `zero2disc` | `zero2pyld` | `pyld2zero` | `zbtprice` | `zbtyield`

Topics

“Term Structure of Interest Rates” on page 2-29
“Fixed-Income Terminology” on page 2-15

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. Use the function `datestr` to convert serial date numbers to formatted date character vectors. `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 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 = [datetime(2000,11,6)
              datetime(2000,12,11)
              datetime(2001,1,15)
              datetime(2001,2,5)
              datetime(2001,3,4)
              datetime(2001,4,2)
              datetime(2001,4,30)
              datetime(2001,6,25)
              datetime(2001,9,4)
              datetime(2001,11,12)];

Settle = datetime(2000,11,3);
InputCompounding = 12;
InputBasis = 2;
OutputCompounding = 1;
OutputBasis = 2;

[ParRates, CurveDates] = zero2pyld(ZeroRates, CurveDates,...
Settle, 'InputCompounding',12,'InputBasis',2,'OutputCompounding',1,'OutputBasis',2)
```

```
ParRates = 10×1

   -0.0436
    0.0611
    0.0579
    0.0567
    0.0550
    0.0543
    0.0541
    0.0546
    0.0565
    0.0561


CurveDates = 10x1 datetime
   06-Nov-2000
   11-Dec-2000
   15-Jan-2001
   05-Feb-2001
   04-Mar-2001
   02-Apr-2001
   30-Apr-2001
   25-Jun-2001
   04-Sep-2001
   12-Nov-2001
```

Demonstrate a Roundtrip From `zero2pyld` to `pyld2zero`

Given the following zero curve and its maturity dates, return the `ParRates`.

```
Settle = datetime(2013,2,1);

CurveDates = [datetime(2014,2,1)
             datetime(2015,2,1)
             datetime(2016,2,1)
             datetime(2018,2,1)
             datetime(2020,2,1)
             datetime(2023,2,1)
             datetime(2033,2,1)
             datetime(2043,2,1) ];
```

```
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 = 8×1

    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 = 8×1

    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
```

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

datetime array | string array | date character vector

Maturity dates which correspond to the input `ZeroRates`, specified as a `NUMBONDS`-by-`1` vector using a datetime array, string array, or date character vectors.

To support existing code, `zero2pyld` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `datetime` | `string` | `char`

`Settle` — Common settlement date for `ZeroRates`

datetime scalar | string scalar | date character vector

Common settlement date for input `ZeroRates`, specified as a scalar datetime, string, or date character vector.

To support existing code, `zero2pyld` also accepts serial date numbers as inputs, but they are not recommended.

Data Types: `datetime` | `string` | `char`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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 as the comma-separated pair consisting of `'OutputCompounding'` and 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 as the comma-separated pair consisting of 'OutputBasis' and 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 2-16.

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 as the comma-separated pair consisting of 'InputCompounding' and 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 as the comma-separated pair consisting of 'InputBasis' and 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 2-16.

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

datetime | serial date number

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.

Version History

Introduced before R2006a

R2022b: Serial date numbers not recommended

Not recommended starting in R2022b

Although zero2pyld supports serial date numbers, datetime values are recommended instead. The datetime data type provides flexible date and time formats, storage out to nanosecond precision, and properties to account for time zones and daylight saving time.

To convert serial date numbers or text to datetime values, use the datetime function. For example:

```
t = datetime(738427.656845093,"ConvertFrom","datenum");
y = year(t)
```

```
y =

        2021
```

There are no plans to remove support for serial date number inputs.

See Also

fwd2zero | zero2fwd | getForwardRates | datetime | disc2zero | zero2disc | pyld2zero | zbtprice | zbtyield | datetime

Topics

"Term Structure of Interest Rates" on page 2-29
"Fixed-Income Terminology" on page 2-15

compact

Create compact credit scorecard

Syntax

```
csc = compact(sc)
```

Description

`csc = compact(sc)` converts an existing `creditscorecard` object to a `compactCreditScorecard` object (`csc`).

Note To use this function, you must have a license for Risk Management Toolbox.

Examples

Convert `creditscorecard` Object into `compactCreditScorecard` Object

Convert a `creditscorecard` object into a `compactCreditScorecard` object to use `displaypoints` (Risk Management Toolbox), `score` (Risk Management Toolbox), and `probdefault` (Risk Management Toolbox) from Risk Management Toolbox™ with the object.

First, create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011).

```
load CreditCardData.mat
sc = creditscorecard(data)
```

```
sc = 
  creditscorecard with properties:

                GoodLabel: 0
              ResponseVar: 'status'
               WeightsVar: ''
                 VarNames: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustI
        NumericPredictors: {'CustID'  'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBala
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
           BinMissingData: 0
                    IDVar: ''
            PredictorVars: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustI
                     Data: [1200x11 table]
```

Before creating a `compactCreditScorecard` object, you must use `autobinning` and `fitmodel` with the `creditscorecard` object.

```
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 + AMBala
    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
```

Convert the `creditscorecard` object into a `compactCreditScorecard` object by using the `compact` function. To use `compact`, you must have a Risk Management Toolbox™ license.

```
csc = compact(sc)
```

```
csc = 
  compactCreditScorecard with properties:

              Description: ''
                GoodLabel: 0
              ResponseVar: 'status'
               WeightsVar: ''
        NumericPredictors: {'CustAge'  'CustIncome'  'TmWBank'  'AMBalance'}
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
            PredictorVars: {'CustAge'  'ResStatus'  'EmpStatus'  'CustIncome'  'TmWBank'  'OtherC
```

You can then use `displaypoints` (Risk Management Toolbox), `score` (Risk Management Toolbox), and `probdefault` (Risk Management Toolbox) from Risk Management Toolbox™ with the `compactCreditScorecard` object.

Input Arguments

`sc` — Credit scorecard model

`creditscorecard` object

Credit scorecard model, specified as a `creditscorecard` object.

Note You must use a creditscorecard object (sc) for input that has been previously processed with autobinning and fitmodel, or fitConstrainedModel. Optionally, you can use formatpoints in addition to these functions.

Output Arguments

csc — Compact credit scorecard

compactCreditScorecard object

Compact credit scorecard, returned as a compactCreditScorecard object. You can then use displaypoints, score, and probdefault from the Risk Management Toolbox with this csc object.

Version History

Introduced in R2019a

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

creditscorecard | compactCreditScorecard | autobinning | bininfo | predictorinfo | modifypredictor | plotbins | modifybins | bindata | fitmodel | formatpoints | displaypoints | setmodel | probdefault | validatemodel | table

Topics

“Case Study for Credit Scorecard Analysis” on page 8-70
“Credit Scorecard Modeling with Missing Values” on page 8-56
“compactCreditScorecard Object Workflow” (Risk Management Toolbox)
“Troubleshooting Credit Scorecard Results” on page 8-63
“Credit Scorecard Modeling Workflow” on page 8-51
“About Credit Scorecards” on page 8-47

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-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 + AMBala
    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
```

Scores for Missing or Out-of-Range Data When Using the 'BinMissingData' Option

This example describes the assignment of points for missing data when the `'BinMissingData'` option is set to `true`.

- Predictors that have missing data in the training set have an explicit bin for `<missing>` with corresponding points in the final scorecard. These points are computed from the Weight-of-Evidence (WOE) value for the `<missing>` bin and the logistic model coefficients. For scoring purposes, these points are assigned to missing values and to out-of-range values.

- Predictors with no missing data in the training set have no <missing> bin, therefore no WOE can be estimated from the training data. By default, the points for missing and out-of-range values are set to NaN, and this leads to a score of NaN when running score. For predictors that have no explicit <missing> bin, use the name-value argument 'Missing' in formatpoints to indicate how missing data should be treated for scoring purposes.

Create a creditscorecard object using the CreditCardData.mat file to load the dataMissing with missing values.

```
load CreditCardData.mat
head(dataMissing,5)
```

```
    CustID    CustAge    TmAtAddress     ResStatus     EmpStatus    CustIncome    TmWBank    Othe
    ______    _______    ___________    ___________    _________    __________    _______    ____

      1          53          62         <undefined>    Unknown        50000         55         Ye
      2          61          22         Home Owner     Employed       52000         25         Ye
      3          47          30         Tenant         Employed       37000         61         No
      4         NaN          75         Home Owner     Employed       53000         20         Ye
      5          68          56         Home Owner     Employed       53000         14         Ye
```

```
fprintf('Number of rows: %d\n',height(dataMissing))
```

```
Number of rows: 1200
```

```
fprintf('Number of missing values CustAge: %d\n',sum(ismissing(dataMissing.CustAge)))
```

```
Number of missing values CustAge: 30
```

```
fprintf('Number of missing values ResStatus: %d\n',sum(ismissing(dataMissing.ResStatus)))
```

```
Number of missing values ResStatus: 40
```

Use creditscorecard with the name-value argument 'BinMissingData' set to true to bin the missing numeric or categorical data in a separate bin. Apply automatic binning.

```
sc = creditscorecard(dataMissing,'IDVar','CustID','BinMissingData',true);
sc = autobinning(sc);

disp(sc)
```

```
  creditscorecard with properties:

                GoodLabel: 0
              ResponseVar: 'status'
               WeightsVar: ''
                 VarNames: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIn
        NumericPredictors: {'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBalance'  'Uti
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
           BinMissingData: 1
                    IDVar: 'CustID'
            PredictorVars: {'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIncome'  'Tr
                     Data: [1200x11 table]
```

Set a minimum value of zero for CustAge and CustIncome. With this, any negative age or income information becomes invalid or "out-of-range". For scoring purposes, out-of-range values are given the same points as missing values.

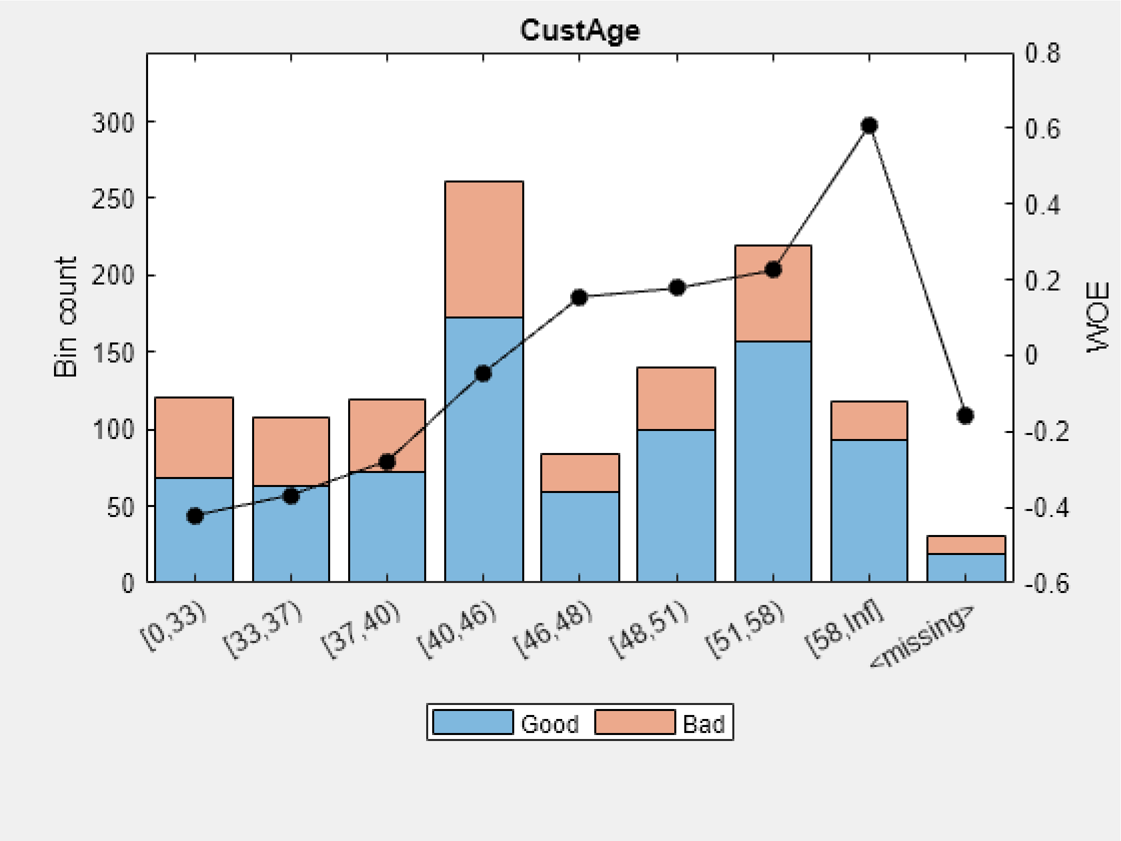
```
sc = modifybins(sc,'CustAge','MinValue',0);
sc = modifybins(sc,'CustIncome','MinValue',0);
```

Display and plot bin information for numeric data for `'CustAge'` that includes missing data in a separate bin labelled `<missing>`.

```
[bi,cp] = bininfo(sc,'CustAge');
disp(bi)
```

```
         Bin           Good    Bad     Odds        WOE       InfoValue 
    _____________    ____    ___    ______    ________    __________

    {'[0,33)'   }     69     52     1.3269    -0.42156      0.018993
    {'[33,37)'  }     63     45        1.4    -0.36795      0.012839
    {'[37,40)'  }     72     47     1.5319     -0.2779     0.0079824
    {'[40,46)'  }    172     89     1.9326    -0.04556     0.0004549
    {'[46,48)'  }     59     25       2.36     0.15424     0.0016199
    {'[48,51)'  }     99     41     2.4146     0.17713     0.0035449
    {'[51,58)'  }    157     62     2.5323     0.22469     0.0088407
    {'[58,Inf]' }     93     25       3.72     0.60931      0.032198
    {'<missing>'}     19     11     1.7273    -0.15787    0.00063885
    {'Totals'   }    803    397     2.0227         NaN      0.087112
```

```
plotbins(sc,'CustAge')
```

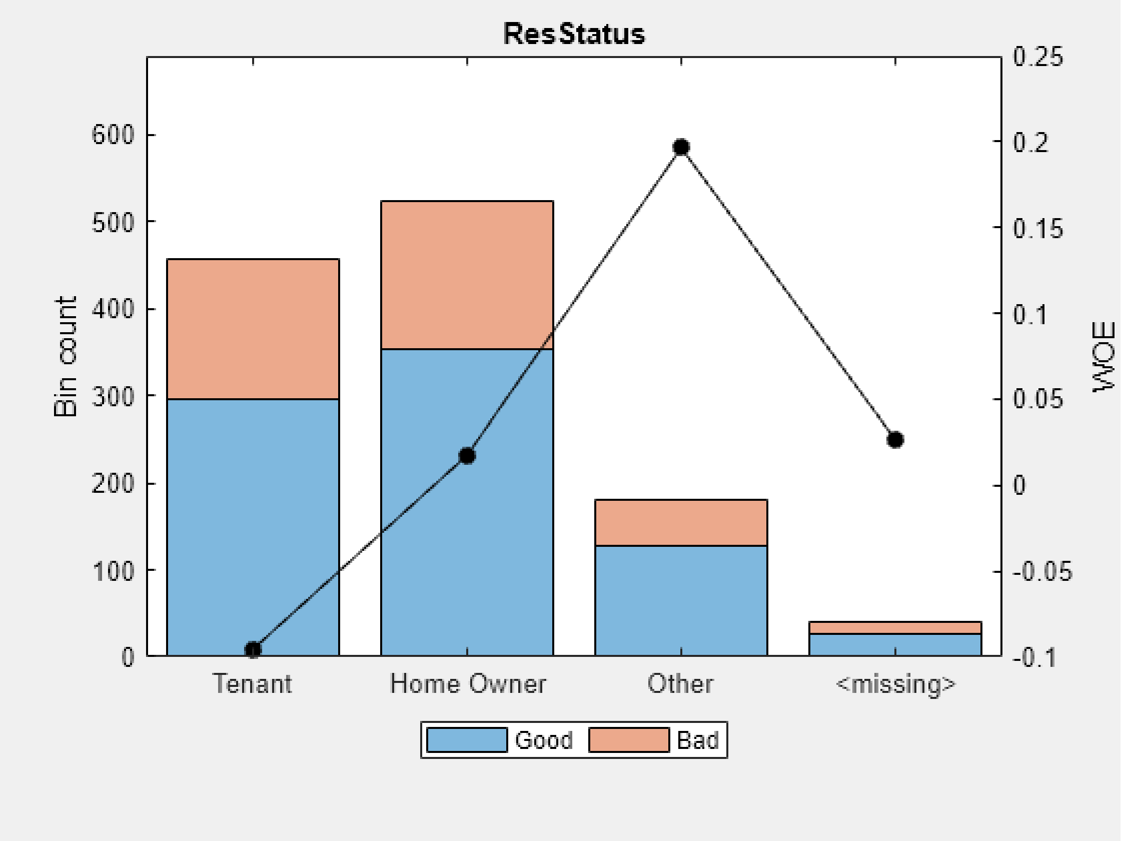


Display and plot bin information for categorical data for `'ResStatus'` that includes missing data in a separate bin labelled `<missing>`.

```
[bi,cg] = bininfo(sc,'ResStatus');
disp(bi)
```

```
        Bin             Good    Bad     Odds         WOE        InfoValue 
    ______________      ____    ___    ______    _________    __________

    {'Tenant'    }      296     161    1.8385    -0.095463     0.0035249
    {'Home Owner'}      352     171    2.0585     0.017549    0.00013382
    {'Other'     }      128      52    2.4615      0.19637     0.0055808
    {'<missing>' }       27      13    2.0769     0.026469    2.3248e-05
    {'Totals'    }      803     397    2.0227          NaN     0.0092627
```

```
plotbins(sc,'ResStatus')
```



For the 'CustAge' and 'ResStatus' predictors, there is missing data (NaNs and <undefined>) in the training data, and the binning process estimates a WOE value of -0.15787 and 0.026469 respectively for missing data in these predictors, as shown above.

For EmpStatus and CustIncome there is no explicit bin for missing values because the training data has no missing values for these predictors.

```
bi = bininfo(sc,'EmpStatus');
disp(bi)
```

```
        Bin            Good    Bad     Odds        WOE       InfoValue
    ____________       ____    ___    ______    ________    _________

    {'Unknown' }       396     239    1.6569    -0.19947     0.021715
    {'Employed'}    407     158    2.5759      0.2418    0.026323
    {'Totals'  }    803     397    2.0227         NaN    0.048038
```

```
bi = bininfo(sc,'CustIncome');
disp(bi)
```

```
           Bin            Good    Bad     Odds         WOE       InfoValue 
    _________________    ____    ___    _______    _________    __________

    {'[0,29000)'    }     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 `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). For predictors that have missing data, there is an explicit `<missing>` bin, with a corresponding WOE value computed from the data. When using `fitmodel`, the corresponding WOE value for the `<missing>` bin is applied when performing the WOE transformation.

```
[sc,mdl] = 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 = 1442.8477, Chi2Stat = 4.4974731, PValue = 0.033944979
6. Adding ResStatus, Deviance = 1438.9783, Chi2Stat = 3.86941, PValue = 0.049173805
7. Adding OtherCC, Deviance = 1434.9751, Chi2Stat = 4.0031966, PValue = 0.045414057

Generalized linear regression model:
    logit(status) ~ 1 + CustAge + ResStatus + EmpStatus + CustIncome + TmWBank + OtherCC + AMBala
    Distribution = Binomial

Estimated Coefficients:
                   Estimate       SE       tStat       pValue  
                   ________    ________    ______    __________

    (Intercept)    0.70229     0.063959     10.98    4.7498e-28
    CustAge        0.57421      0.25708    2.2335      0.025513
    ResStatus       1.3629      0.66952    2.0356       0.04179
    EmpStatus      0.88373       0.2929    3.0172      0.002551
    CustIncome     0.73535       0.2159     3.406    0.00065929
    TmWBank         1.1065      0.23267    4.7556    1.9783e-06
    OtherCC         1.0648      0.52826    2.0156      0.043841
    AMBalance       1.0446      0.32197    3.2443     0.0011775


1200 observations, 1192 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 88.5, p-value = 2.55e-16
```

Scale the scorecard points by the "points, odds, and points to double the odds (PDO)" method using the `'PointsOddsAndPDO'` argument of `formatpoints`. 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).

Display the scorecard showing the scaled points for predictors retained in the fitting model.

```
sc = formatpoints(sc,'PointsOddsAndPDO',[500 2 50]);
PointsInfo = displaypoints(sc)
```

```
PointsInfo=38×3 table
     Predictors           Bin          Points
    _____________    ______________    ______

    {'CustAge'  }    {'[0,33)'    }    54.062
    {'CustAge'  }    {'[33,37)'   }    56.282
    {'CustAge'  }    {'[37,40)'   }    60.012
    {'CustAge'  }    {'[40,46)'   }    69.636
    {'CustAge'  }    {'[46,48)'   }    77.912
    {'CustAge'  }    {'[48,51)'   }     78.86
    {'CustAge'  }    {'[51,58)'   }     80.83
    {'CustAge'  }    {'[58,Inf]'  }     96.76
    {'CustAge'  }    {'<missing>' }    64.984
    {'ResStatus'}    {'Tenant'    }    62.138
    {'ResStatus'}    {'Home Owner'}    73.248
    {'ResStatus'}    {'Other'     }    90.828
    {'ResStatus'}    {'<missing>' }    74.125
    {'EmpStatus'}    {'Unknown'   }    58.807
    {'EmpStatus'}    {'Employed'  }    86.937
    {'EmpStatus'}    {'<missing>' }       NaN
      ⋮
```

Notice that points for the `<missing>` bin for `CustAge` and `ResStatus` are explicitly shown (as `64.9836` and `74.1250`, respectively). These points are computed from the WOE value for the `<missing>` bin, and the logistic model coefficients.

For predictors that have no missing data in the training set, there is no explicit `<missing>` bin. By default the points are set to `NaN` for missing data and they lead to a score of `NaN` when running `score`. For predictors that have no explicit `<missing>` bin, use the name-value argument `'Missing'` in `formatpoints` to indicate how missing data should be treated for scoring purposes.

For the purpose of illustration, take a few rows from the original data as test data and introduce some missing data. Also introduce some invalid, or out-of-range values. For numeric data, values below the minimum (or above the maximum) allowed are considered invalid, such as a negative value for age (recall `'MinValue'` was earlier set to `0` for `CustAge` and `CustIncome`). For categorical data, invalid values are categories not explicitly included in the scorecard, for example, a residential status not previously mapped to scorecard categories, such as "House", or a meaningless string such as "abc123".

```
tdata = dataMissing(11:18,mdl.PredictorNames); % Keep only the predictors retained in the model
% Set some missing values
tdata.CustAge(1) = NaN;
tdata.ResStatus(2) = '<undefined>';
tdata.EmpStatus(3) = '<undefined>';
tdata.CustIncome(4) = NaN;
% Set some invalid values
```

```
tdata.CustAge(5) = -100;
tdata.ResStatus(6) = 'House';
tdata.EmpStatus(7) = 'Freelancer';
tdata.CustIncome(8) = -1;
disp(tdata)
```

```
    CustAge     ResStatus      EmpStatus      CustIncome    TmWBank    OtherCC    AMBalance
    _______    ___________    ___________    __________    _______    _______    _________

      NaN      Tenant         Unknown           34000         44        Yes         119.8
       48      <undefined>    Unknown           44000         14        Yes        403.62
       65      Home Owner     <undefined>       48000          6        No         111.88
       44      Other          Unknown             NaN         35        No         436.41
     -100      Other          Employed          46000         16        Yes        162.21
       33      House          Employed          36000         36        Yes        845.02
       39      Tenant         Freelancer        34000         40        Yes        756.26
       24      Home Owner     Employed             -1         19        Yes        449.61
```

Score the new data and see how points are assigned for missing `CustAge` and `ResStatus`, because we have an explicit bin with points for `<missing>`. However, for `EmpStatus` and `CustIncome` the `score` function sets the points to `NaN`.

```
[Scores,Points] = score(sc,tdata);
disp(Scores)
```

```
  481.2231
  520.8353
       NaN
       NaN
  551.7922
  487.9588
       NaN
       NaN
```

```
disp(Points)
```

```
    CustAge    ResStatus    EmpStatus    CustIncome    TmWBank    OtherCC    AMBalance
    _______    _________    _________    __________    _______    _______    _________

    64.984      62.138       58.807        67.893      61.858     75.622      89.922
     78.86      74.125       58.807        82.439      61.061     75.622      89.922
     96.76      73.248          NaN        96.969      51.132     50.914      89.922
    69.636      90.828       58.807           NaN      61.858     50.914      89.922
    64.984      90.828       86.937        82.439      61.061     75.622      89.922
    56.282      74.125       86.937        70.107      61.858     75.622      63.028
    60.012      62.138          NaN        67.893      61.858     75.622      63.028
    54.062      73.248       86.937           NaN      61.061     75.622      89.922
```

Use the name-value argument `'Missing'` in `formatpoints` to choose how to assign points to missing values for predictors that do not have an explicit `<missing>` bin. In this example, use the `'MinPoints'` option for the `'Missing'` argument. The minimum points for `EmpStatus` in the scorecard displayed above are `58.8072`, and for `CustIncome` the minimum points are `29.3753`.

```
sc = formatpoints(sc,'Missing','MinPoints');
[Scores,Points] = score(sc,tdata);
disp(Scores)
```

```
  481.2231
  520.8353
  517.7532
  451.3405
  551.7922
  487.9588
  449.3577
  470.2267
```

```
disp(Points)
```

```
    CustAge    ResStatus    EmpStatus    CustIncome    TmWBank    OtherCC    AMBalance
    _______    _________    _________    __________    _______    _______    _________

    64.984      62.138       58.807        67.893      61.858     75.622      89.922  
     78.86      74.125       58.807        82.439      61.061     75.622      89.922  
     96.76      73.248       58.807        96.969      51.132     50.914      89.922  
    69.636      90.828       58.807        29.375      61.858     50.914      89.922  
    64.984      90.828       86.937        82.439      61.061     75.622      89.922  
    56.282      74.125       86.937        70.107      61.858     75.622      63.028  
    60.012      62.138       58.807        67.893      61.858     75.622      63.028  
    54.062      73.248       86.937        29.375      61.061     75.622      89.922  
```

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-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 + AMBala
    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 = 11×1

    0.8252
    0.6553
    1.2443
    0.9478
    0.5690
    1.6192
    0.4899
    0.3824
    0.2945
    1.4401
      ⋮
```

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`.

Version History

Introduced in R2014b

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

`creditscorecard` | `autobinning` | `bininfo` | `predictorinfo` | `modifypredictor` | `plotbins` | `modifybins` | `bindata` | `fitmodel` | `formatpoints` | `displaypoints` | `setmodel` | `probdefault` | `validatemodel` | `table`

Topics

“Case Study for Credit Scorecard Analysis” on page 8-70
“Credit Scorecard Modeling with Missing Values” on page 8-56
“Troubleshooting Credit Scorecard Results” on page 8-63
“Credit Scorecard Modeling Workflow” on page 8-51
“About Credit Scorecards” on page 8-47

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 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-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 + AMBala
    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=37×3 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
    {'CustAge'   }    {'<missing>'   }          NaN
    {'ResStatus' }    {'Tenant'      }    -0.031252
    {'ResStatus' }    {'Home Owner'  }      0.12696
    {'ResStatus' }    {'Other'       }      0.37641
    {'ResStatus' }    {'<missing>'   }          NaN
    {'EmpStatus' }    {'Unknown'     }    -0.076317
    {'EmpStatus' }    {'Employed'    }      0.31449
    {'EmpStatus' }    {'<missing>'   }          NaN
    {'CustIncome'}    {'[-Inf,29000)'}     -0.45716
      ⋮


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=37×3 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
    {'CustAge'   }    {'<missing>'   }       NaN
    {'ResStatus' }    {'Tenant'      }    62.031
    {'ResStatus' }    {'Home Owner'  }    73.444
    {'ResStatus' }    {'Other'       }    91.438
    {'ResStatus' }    {'<missing>'   }       NaN
    {'EmpStatus' }    {'Unknown'     }    58.781
    {'EmpStatus' }    {'Employed'    }    86.971
    {'EmpStatus' }    {'<missing>'   }       NaN
    {'CustIncome'}    {'[-Inf,29000)'}    31.309
      ⋮


MinScore = 355.5051

MaxScore = 671.6403
```

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-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 + AMBala
    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=37×3 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
    {'CustAge'   }    {'<missing>'   }          NaN
    {'ResStatus' }    {'Tenant'      }    -0.031252
    {'ResStatus' }    {'Home Owner'  }      0.12696
    {'ResStatus' }    {'Other'       }      0.37641
    {'ResStatus' }    {'<missing>'   }          NaN
    {'EmpStatus' }    {'Unknown'     }    -0.076317
    {'EmpStatus' }    {'Employed'    }      0.31449
    {'EmpStatus' }    {'<missing>'   }          NaN
    {'CustIncome'}    {'[-Inf,29000)'}     -0.45716
      ⋮


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=37×3 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
    {'CustAge'   }    {'<missing>'   }       NaN
    {'ResStatus' }    {'Tenant'      }    62.421
    {'ResStatus' }    {'Home Owner'  }    82.276
    {'ResStatus' }    {'Other'       }    113.58
    {'ResStatus' }    {'<missing>'   }       NaN
    {'EmpStatus' }    {'Unknown'     }    56.765
    {'EmpStatus' }    {'Employed'    }    105.81
    {'EmpStatus' }    {'<missing>'   }       NaN
    {'CustIncome'}    {'[-Inf,29000)'}    8.9706
      ⋮

MinScore = 300.0000

MaxScore = 850
```

As expected, the values of `MinScore` and `MaxScore` correspond to the desired worst and best scores.

Scale Points Using Shift and Slope

This example shows how to use `formatpoints` to scale by providing the `Shift` and `Slope` values. By using `formatpoints` to scale, you can put points and scores in a desired range that is more meaningful for practical purposes. Technically, this involves a linear transformation from the unscaled to the scaled points by the `formatpoints` function.

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011). Use the `'IDVar'` argument in `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-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 + AMBala
    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=37×3 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
    {'CustAge'   }    {'<missing>'   }          NaN
    {'ResStatus' }    {'Tenant'      }    -0.031252
    {'ResStatus' }    {'Home Owner'  }      0.12696
    {'ResStatus' }    {'Other'       }      0.37641
    {'ResStatus' }    {'<missing>'   }          NaN
    {'EmpStatus' }    {'Unknown'     }    -0.076317
    {'EmpStatus' }    {'Employed'    }      0.31449
    {'EmpStatus' }    {'<missing>'   }          NaN
    {'CustIncome'}    {'[-Inf,29000)'}     -0.45716
      ⋮


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=37×3 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
    {'CustAge'   }    {'<missing>'   }       NaN
    {'ResStatus' }    {'Tenant'      }     42.67
    {'ResStatus' }    {'Home Owner'  }    43.619
    {'ResStatus' }    {'Other'       }    45.116
    {'ResStatus' }    {'<missing>'   }       NaN
    {'EmpStatus' }    {'Unknown'     }    42.399
    {'EmpStatus' }    {'Employed'    }    44.744
    {'EmpStatus' }    {'<missing>'   }       NaN
    {'CustIncome'}    {'[-Inf,29000)'}    40.114
      ⋮

MinScore = 292.1401

MaxScore = 318.4355
```

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-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 + AMBala
    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=37×3 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
    {'CustAge'   }    {'<missing>'   }          NaN
    {'ResStatus' }    {'Tenant'      }    -0.031252
    {'ResStatus' }    {'Home Owner'  }      0.12696
    {'ResStatus' }    {'Other'       }      0.37641
    {'ResStatus' }    {'<missing>'   }          NaN
    {'EmpStatus' }    {'Unknown'     }    -0.076317
    {'EmpStatus' }    {'Employed'    }      0.31449
    {'EmpStatus' }    {'<missing>'   }          NaN
    {'CustIncome'}    {'[-Inf,29000)'}     -0.45716
      ⋮


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=38×3 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
    {'CustAge'   }    {'<missing>' }          NaN
    {'ResStatus' }    {'Tenant'    }     -0.13159
    {'ResStatus' }    {'Home Owner'}     0.026616
    {'ResStatus' }    {'Other'     }      0.27607
    {'ResStatus' }    {'<missing>' }          NaN
    {'EmpStatus' }    {'Unknown'   }     -0.17666
    {'EmpStatus' }    {'Employed'  }      0.21415
    {'EmpStatus' }    {'<missing>' }          NaN
      ⋮


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-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 + AMBala
    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=37×3 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
    {'CustAge'   }    {'<missing>'   }          NaN
    {'ResStatus' }    {'Tenant'      }    -0.031252
    {'ResStatus' }    {'Home Owner'  }      0.12696
    {'ResStatus' }    {'Other'       }      0.37641
    {'ResStatus' }    {'<missing>'   }          NaN
    {'EmpStatus' }    {'Unknown'     }    -0.076317
    {'EmpStatus' }    {'Employed'    }      0.31449
    {'EmpStatus' }    {'<missing>'   }          NaN
    {'CustIncome'}    {'[-Inf,29000)'}     -0.45716
      ⋮


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=37×3 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
    {'CustAge'   }    {'<missing>'   }       NaN
    {'ResStatus' }    {'Tenant'      }    62.421
    {'ResStatus' }    {'Home Owner'  }    82.276
    {'ResStatus' }    {'Other'       }    113.58
    {'ResStatus' }    {'<missing>'   }       NaN
    {'EmpStatus' }    {'Unknown'     }    56.765
    {'EmpStatus' }    {'Employed'    }    105.81
    {'EmpStatus' }    {'<missing>'   }       NaN
    {'CustIncome'}    {'[-Inf,29000)'}    8.9706
      ⋮
```

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=37×3 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
    {'CustAge'   }    {'<missing>'   }     NaN
    {'ResStatus' }    {'Tenant'      }      62
    {'ResStatus' }    {'Home Owner'  }      82
    {'ResStatus' }    {'Other'       }     114
    {'ResStatus' }    {'<missing>'   }     NaN
    {'EmpStatus' }    {'Unknown'     }      57
    {'EmpStatus' }    {'Employed'    }     106
    {'EmpStatus' }    {'<missing>'   }     NaN
    {'CustIncome'}    {'[-Inf,29000)'}       9
      ⋮
```

Rounding and Default Probabilities

This example shows that rounding scorecard points can modify the original risk ranking of a credit scorecard. You can control rounding by using `formatpoints` with the optional name-value pair argument for `'Rounding'`.

Credit scores rank customers by risk. If higher scores are given to better, less risky customers, then higher scores must correspond to lower default probabilities. When you use the name-value pair argument for `'Rounding'`, depending on the value for `'Rounding'`, the rounding behavior is:

- When `'Rounding'` is set to `'None'` (default option), no rounding is applied to points or scores, and the risk ranking is completely consistent with the calibrated model.
- When `'Rounding'` is set to `'FinalScore'`, rounding is only applied to the final scores. In this case: a) Customers with different scores (different risk) may have the same rounded score. b) Customers with the same rounded score may have different default probabilities. c) Customer with higher rounded scores will always have lower default probability than customers with lower scores.
- When `'Rounding'` is set to `'AllPoints'`, rounding is applied to all points in the scorecard (all bins, all predictors). In this case: a) Customers with different scores (different risk) may have the same rounded score, or their ranking may even be reversed (the customer with the lower original score may have a higher rounded score). b) Customers with the same rounded score may have different default probabilities. c) Customer with higher rounded scores may in some cases have *higher* default probabilities than customers with lower scores.

Create a `creditscorecard`

To demonstrate the rounding behavior, first create a `creditscorecard` object.

```
load CreditCardData
sc = creditscorecard(data,'IDVar','CustID','ResponseVar','status');
sc = autobinning(sc);
sc = modifybins(sc,'CustIncome','CutPoints',20000:5000:60000);
sc = fitmodel(sc);
```

```
1. Adding CustIncome, Deviance = 1487.9719, Chi2Stat = 35.469392, PValue = 2.5909009e-09
2. Adding TmWBank, Deviance = 1465.7998, Chi2Stat = 22.172089, PValue = 2.4927133e-06
3. Adding AMBalance, Deviance = 1455.206, Chi2Stat = 10.593833, PValue = 0.0011346548
4. Adding EmpStatus, Deviance = 1446.3918, Chi2Stat = 8.8142314, PValue = 0.0029889009
5. Adding CustAge, Deviance = 1440.6825, Chi2Stat = 5.709236, PValue = 0.016875883
6. Adding ResStatus, Deviance = 1436.1363, Chi2Stat = 4.5462043, PValue = 0.032991806
7. Adding OtherCC, Deviance = 1431.9546, Chi2Stat = 4.1817827, PValue = 0.040860699

Generalized linear regression model:
    logit(status) ~ 1 + CustAge + ResStatus + EmpStatus + CustIncome + TmWBank + OtherCC + AMBala
    Distribution = Binomial

Estimated Coefficients:
                   Estimate       SE        tStat       pValue  
                   ________    ________    ______    __________

    (Intercept)    0.70247     0.064046    10.968    5.4345e-28
    CustAge        0.60579      0.24405    2.4822      0.013058
    ResStatus       1.4463      0.65427    2.2105       0.02707
    EmpStatus      0.90501      0.29262    3.0928     0.0019828
    CustIncome     0.70869      0.20535    3.4512    0.00055815
    TmWBank         1.0839      0.23244    4.6631    3.1145e-06
    OtherCC         1.0906      0.52936    2.0602      0.039377
    AMBalance       1.0148      0.32273    3.1445     0.0016636


1200 observations, 1192 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 91.5, p-value = 6.12e-17
```

Apply the 'Rounding' Options

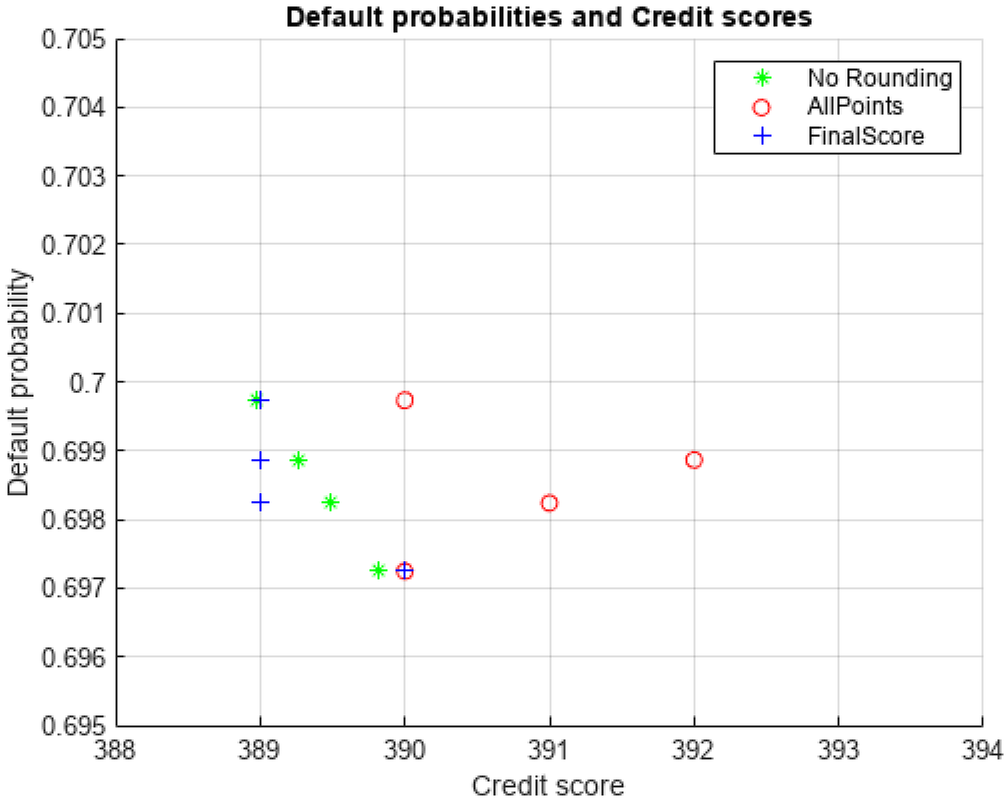
Apply each of the three 'Rounding' options to the creditscorecard object.

```
sc = formatpoints(sc,'PointsOddsAndPDO',[500 2 50]); % No Rounding
points1  = displaypoints(sc);
[S1,P1]  = score(sc);
defProb1 = probdefault(sc);

sc = formatpoints(sc,'Round','AllPoints'); % 'AllPoints' Rounding
points2  = displaypoints(sc);
[S2,P2]  = score(sc);
defProb2 = probdefault(sc);

sc = formatpoints(sc,'Round','FinalScore'); % 'FinalScore' Rounding
points3  = displaypoints(sc);
[S3,P3]  = score(sc);
defProb3 = probdefault(sc);
```

Compare the 'Rounding' Options

Visualize the default probabilities versus the scores.

```
figure
hold on
scatter(S1, defProb1, 'g*')
scatter(S2, defProb2, 'ro')
scatter(S3, defProb3, 'b+')
legend('No Rounding','AllPoints','FinalScore')
axis([388 394 0.695 0.705])
xlabel('Credit score')
ylabel('Default probability')
title('Default probabilities and Credit scores')
grid
```



Inspect the points and total scores for each `'Rounding'` option, in table format.

```
ind = [208 363 694 886];
ProbDefault = defProb1(ind)
```

```
ProbDefault = 4×1

    0.6997
    0.6989
    0.6982
    0.6972
```

```
% ScoreNoRounding = S1(ind)
PointsNoRounding = P1(ind,:);
PointsNoRounding.Total = S1(ind)
```

```
PointsNoRounding=4×8 table
    CustAge    ResStatus    EmpStatus    CustIncome    TmWBank    OtherCC    AMBalance    Total 
    _______    _________    _________    __________    _______    _______    _________    ______

      52.9      61.555       58.503        24.647      51.551     50.416        89.4      388.97
     67.65      61.555       58.503        24.647      51.551     75.723       49.64      389.27
    54.234      61.555       58.503        24.647      51.551     75.723      63.271      389.48
      52.9      92.441       58.503        24.647      51.277     50.416       49.64      389.82
```

```
% ScoreAllPoints = S2(ind)
PointsAllPoints = P2(ind,:);
PointsAllPoints.Total = S2(ind)

PointsAllPoints=4×8 table
    CustAge    ResStatus    EmpStatus    CustIncome    TmWBank    OtherCC    AMBalance    Total
    _______    _________    _________    __________    _______    _______    _________    _____

      53          62           59            25           52         50          89        390
      68          62           59            25           52         76          50        392
      54          62           59            25           52         76          63        391
      53          92           59            25           61         50          50        390
```

```
% ScoreFinalScore = S3(ind)
PointsFinalScore = P3(ind,:);
PointsFinalScore.Total = S3(ind)

PointsFinalScore=4×8 table
    CustAge    ResStatus    EmpStatus    CustIncome    TmWBank    OtherCC    AMBalance    Total
    _______    _________    _________    __________    _______    _______    _________    _____

      52.9      61.555       58.503        24.647      51.551     50.416        89.4       389
     67.65      61.555       58.503        24.647      51.551     75.723       49.64       389
    54.234      61.555       58.503        24.647      51.551     75.723      63.271       389
      52.9      92.441       58.503        24.647      61.277     50.416       49.64       390
```

The original `creditscorecard` model, without rounding, was calibrated to the data with a logistic regression. The ranking and probabilities have a statistical foundation.

Rounding, however, effectively modifies the `creditscorecard` model. When only the final score is rounded, this leads to some "ties" in rounded scores, but at least the risk ranking *across* scores is preserved (because if *s1* <= *s2*, then round(*s1*) <= round(*s2*)).

However, when you round all points, a score may gain extra points by chance. For example, in the second row in the table (row 363 of original data), the points for all predictors are rounded up by almost `0.5`. The original score is `389.27`. Rounding the final score makes it `389`. However, rounding all points makes it `392`, that is three points higher than rounding the final score.

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);
```

Indicate that the minimum allowed value for `'CustAge'` is zero. This makes any negative values for age invalid or out-of-range.

```
sc = modifybins(sc,'CustAge','MinValue',0);
```

Fit a linear regression model using default parameters.

```
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 + AMBala
    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 there are missing or out of range observations in the data that you want to score. Notice that by default, the points and score assigned to the missing value is `NaN`.

```
% Set up a data set with missing and out of range data for illustration purposes
newdata = data(1:5,:);
newdata.CustAge(1) = NaN; % missing
newdata.CustAge(2) = -100; % invalid
newdata.ResStatus(3) = '<undefined>'; % missing
newdata.ResStatus(4) = 'House'; % invalid
disp(newdata)
```

```
    CustID    CustAge    TmAtAddress     ResStatus     EmpStatus    CustIncome    TmWBank    Othe
    ______    _______    ___________    ___________    _________    __________    _______    ____

      1         NaN          62         Tenant         Unknown        50000         55         Ye
      2        -100          22         Home Owner     Employed       52000         25         Ye
      3          47           30          <undefined>    Employed       37000           61           No
      4          50           75          House          Employed       53000           20           Ye
      5          68           56          Home Owner     Employed       53000           14           Ye
```

```
[Scores,Points] = score(sc,newdata);
disp(Scores)
```

```
       NaN
       NaN
       NaN
       NaN
    1.4535
```

```
disp(Points)
```

```
    CustAge     ResStatus    EmpStatus    CustIncome     TmWBank     OtherCC     AMBalance
    _______     _________    _________    __________    _________    ________    _________

        NaN     -0.031252    -0.076317      0.43693       0.39607     0.15842    -0.017472
        NaN       0.12696      0.31449      0.43693     -0.033752     0.15842    -0.017472
    0.21445           NaN      0.31449     0.081611       0.39607    -0.19168    -0.017472
    0.23039           NaN      0.31449      0.43693     -0.044811     0.15842      0.35551
      0.479       0.12696      0.31449      0.43693     -0.044811     0.15842    -0.017472
```

Use the name-value pair argument `Missing` to replace `NaN` with points corresponding to a zero Weight-of-Evidence (WOE).

```
sc = formatpoints(sc,'Missing','ZeroWOE');
[Scores,Points] = score(sc,newdata);
disp(Scores)
```

```
    0.9667
    1.0859
    0.8978
    1.5513
    1.4535
```

```
disp(Points)
```

```
    CustAge     ResStatus    EmpStatus    CustIncome     TmWBank     OtherCC     AMBalance
    _______     _________    _________    __________    _________    ________    _________

    0.10034     -0.031252    -0.076317      0.43693       0.39607     0.15842    -0.017472
    0.10034       0.12696      0.31449      0.43693     -0.033752     0.15842    -0.017472
    0.21445       0.10034      0.31449     0.081611       0.39607    -0.19168    -0.017472
    0.23039       0.10034      0.31449      0.43693     -0.044811     0.15842      0.35551
      0.479       0.12696      0.31449      0.43693     -0.044811     0.15842    -0.017472
```

Alternatively, use the name-value pair argument `Missing` to replace the missing value with the minimum points for the predictors that have the missing values.

```
sc = formatpoints(sc,'Missing','MinPoints');
[Scores,Points] = score(sc,newdata);
disp(Scores)
```

```
    0.7074
    0.8266
    0.7662
    1.4197
    1.4535
```

```
disp(Points)
```

```
    CustAge     ResStatus    EmpStatus    CustIncome     TmWBank     OtherCC     AMBalance
    ________    _________    _________    __________    _________    ________    _________

    -0.15894    -0.031252    -0.076317      0.43693       0.39607     0.15842    -0.017472
    -0.15894      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    -0.031252      0.31449      0.43693     -0.044811     0.15842      0.35551
       0.479      0.12696      0.31449      0.43693     -0.044811     0.15842    -0.017472
```

As a third alternative, use the name-value pair argument `Missing` to replace the missing value with the maximum points for the predictors that have the missing values.

```
sc = formatpoints(sc,'Missing','MaxPoints');
[Scores,Points] = score(sc,newdata);
disp(Scores)
```

```
    1.3454
    1.4646
    1.1739
    1.8273
    1.4535
```

```
disp(Points)
```

```
    CustAge    ResStatus    EmpStatus    CustIncome     TmWBank     OtherCC     AMBalance
    _______    _________    _________    __________    _________    ________    _________

      0.479    -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      0.37641      0.31449     0.081611       0.39607    -0.19168    -0.017472
    0.23039      0.37641      0.31449      0.43693     -0.044811     0.15842      0.35551
      0.479      0.12696      0.31449      0.43693     -0.044811     0.15842    -0.017472
```

Verify that the minimum and maximum points assigned to the missing data correspond to the minimum and maximum points for the corresponding predictors. The points for `'CustAge'` are reported in the first seven rows of the points information table. For `'ResStatus'` the points are in rows 8 through 10.

```
PointsInfo = displaypoints(sc);
PointsInfo(1:7,:)
```

```
ans=7×3 table
    Predictors         Bin          Points  
    ___________    ____________    _________

    {'CustAge'}    {'[0,33)'  }     -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
```

```
min(PointsInfo.Points(1:7))
```

```
ans = -0.1589
```

```
max(PointsInfo.Points(1:7))
```

```
ans = 0.4790
```

```
PointsInfo(8:10,:)
```

```
ans=3×3 table
     Predictors           Bin           Points  
    _____________    ______________    _________

    {'CustAge'  }    {'<missing>' }        0.479
    {'ResStatus'}    {'Tenant'    }    -0.031252
    {'ResStatus'}    {'Home Owner'}      0.12696
```

```
min(PointsInfo.Points(8:10))
```

```
ans = -0.0313
```

```
max(PointsInfo.Points(8:10))
```

```
ans = 0.4790
```

Scores for Missing or Out-of-Range Data When Using the 'BinMissingData' Option

This example describes the assignment of points for missing data when the 'BinMissingData' option is set to true.

- Predictors that have missing data in the training set have an explicit bin for <missing> with corresponding points in the final scorecard. These points are computed from the Weight-of-Evidence (WOE) value for the <missing> bin and the logistic model coefficients. For scoring purposes, these points are assigned to missing values and to out-of-range values.
- Predictors with no missing data in the training set have no <missing> bin, therefore no WOE can be estimated from the training data. By default, the points for missing and out-of-range values are set to NaN, and this leads to a score of NaN when running score. For predictors that have no explicit <missing> bin, use the name-value argument 'Missing' in formatpoints to indicate how missing data should be treated for scoring purposes.

Create a creditscorecard object using the CreditCardData.mat file to load the dataMissing with missing values.

```
load CreditCardData.mat
head(dataMissing,5)
```

```
    CustID    CustAge    TmAtAddress     ResStatus     EmpStatus    CustIncome    TmWBank    Othe
    ______    _______    ___________    ___________    _________    __________    _______    ____

      1          53          62         <undefined>    Unknown        50000         55         Ye
      2          61          22         Home Owner     Employed       52000         25         Ye
      3          47          30         Tenant         Employed       37000         61         No
      4         NaN          75         Home Owner     Employed       53000         20         Ye
      5          68          56         Home Owner     Employed       53000         14         Ye
```

```
fprintf('Number of rows: %d\n',height(dataMissing))
```

```
Number of rows: 1200
```

```
fprintf('Number of missing values CustAge: %d\n',sum(ismissing(dataMissing.CustAge)))
```

```
Number of missing values CustAge: 30
```

```
fprintf('Number of missing values ResStatus: %d\n',sum(ismissing(dataMissing.ResStatus)))
```

```
Number of missing values ResStatus: 40
```

Use `creditscorecard` with the name-value argument `'BinMissingData'` set to `true` to bin the missing numeric or categorical data in a separate bin. Apply automatic binning.

```
sc = creditscorecard(dataMissing,'IDVar','CustID','BinMissingData',true);
sc = autobinning(sc);
```

```
disp(sc)
```

```
  creditscorecard with properties:

                GoodLabel: 0
              ResponseVar: 'status'
               WeightsVar: ''
                 VarNames: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIn
        NumericPredictors: {'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBalance'  'Uti
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
           BinMissingData: 1
                    IDVar: 'CustID'
            PredictorVars: {'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIncome'  'Tr
                     Data: [1200x11 table]
```

Set a minimum value of zero for `CustAge` and `CustIncome`. With this, any negative age or income information becomes invalid or "out-of-range". For scoring purposes, out-of-range values are given the same points as missing values.

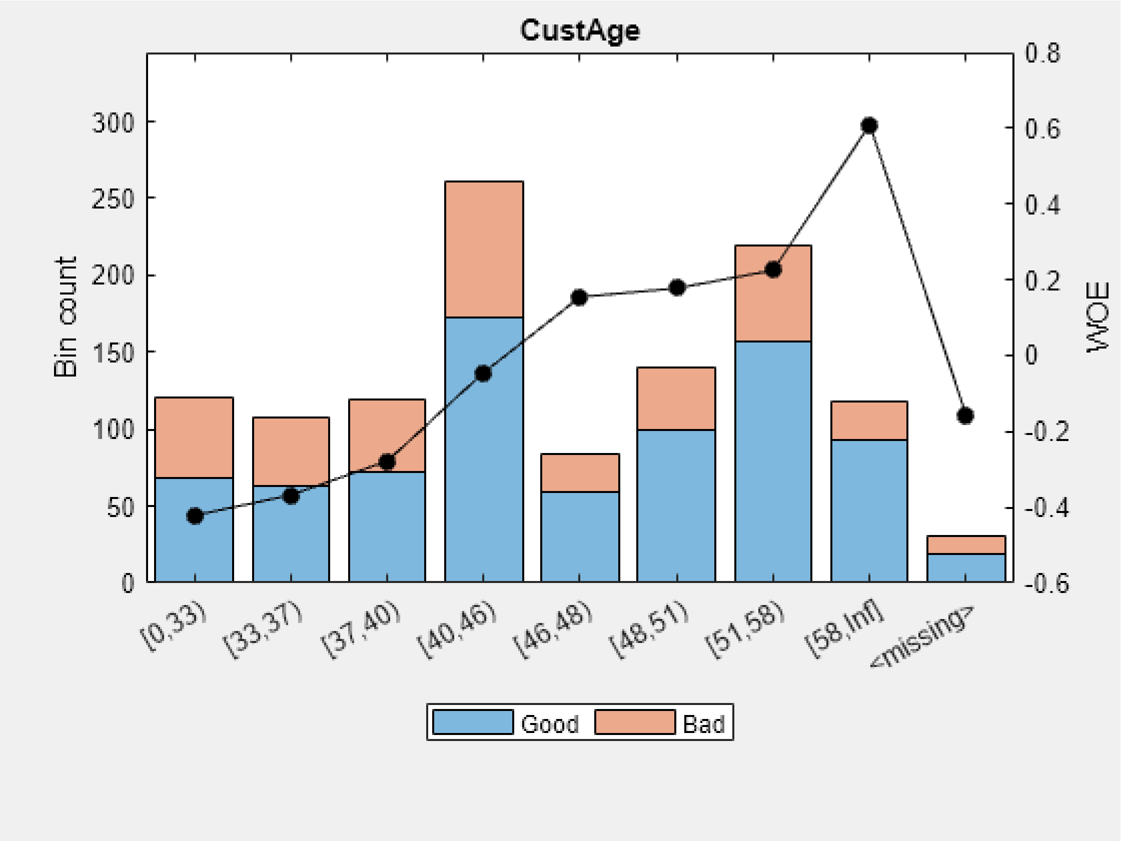
```
sc = modifybins(sc,'CustAge','MinValue',0);
sc = modifybins(sc,'CustIncome','MinValue',0);
```

Display and plot bin information for numeric data for `'CustAge'` that includes missing data in a separate bin labelled `<missing>`.

```
[bi,cp] = bininfo(sc,'CustAge');
disp(bi)
```

```
        Bin          Good    Bad      Odds        WOE       InfoValue 
    _____________    ____    ___    ______    ________    __________

    {'[0,33)'   }     69     52     1.3269    -0.42156      0.018993
    {'[33,37)'  }     63     45        1.4    -0.36795      0.012839
    {'[37,40)'  }     72     47     1.5319     -0.2779     0.0079824
    {'[40,46)'  }    172     89     1.9326    -0.04556     0.0004549
    {'[46,48)'  }     59     25       2.36     0.15424     0.0016199
    {'[48,51)'  }     99     41     2.4146     0.17713     0.0035449
    {'[51,58)'  }    157     62     2.5323     0.22469     0.0088407
    {'[58,Inf]' }     93     25       3.72     0.60931      0.032198
    {'<missing>'}     19     11     1.7273    -0.15787    0.00063885
    {'Totals'   }    803    397     2.0227         NaN      0.087112
```

```
plotbins(sc,'CustAge')
```

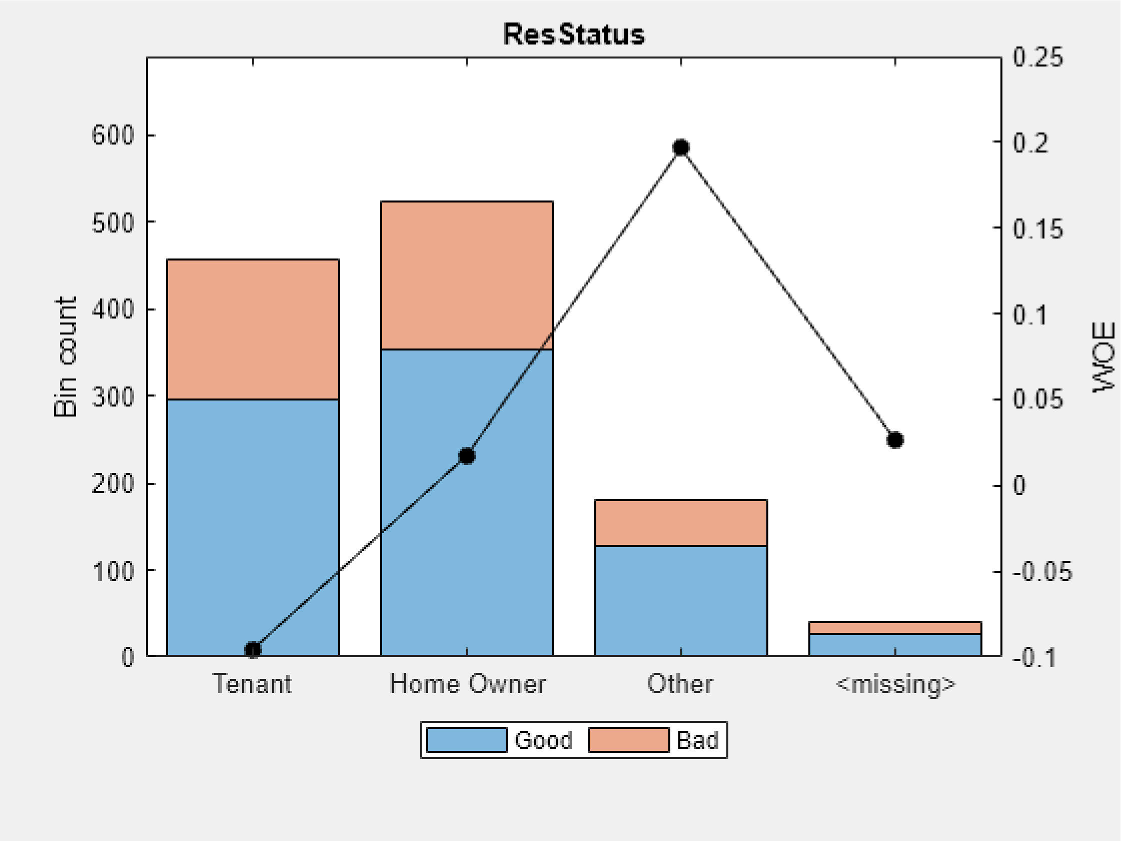


Display and plot bin information for categorical data for `'ResStatus'` that includes missing data in a separate bin labelled `<missing>`.

```
[bi,cg] = bininfo(sc,'ResStatus');
disp(bi)
```

```
         Bin            Good    Bad     Odds        WOE       InfoValue 
    ______________      ____    ___    ______    _________    __________

    {'Tenant'     }     296     161    1.8385    -0.095463     0.0035249
    {'Home Owner'}      352     171    2.0585     0.017549    0.00013382
    {'Other'      }     128      52    2.4615      0.19637     0.0055808
    {'<missing>' }       27      13    2.0769     0.026469    2.3248e-05
    {'Totals'     }     803     397    2.0227          NaN     0.0092627
```

```
plotbins(sc,'ResStatus')
```



For the 'CustAge' and 'ResStatus' predictors, there is missing data (NaNs and <undefined>) in the training data, and the binning process estimates a WOE value of -0.15787 and 0.026469 respectively for missing data in these predictors, as shown above.

For EmpStatus and CustIncome there is no explicit bin for missing values because the training data has no missing values for these predictors.

```
bi = bininfo(sc,'EmpStatus');
disp(bi)
```

```
        Bin           Good    Bad     Odds        WOE       InfoValue
    ____________      ____    ___    ______    ________    _________

    {'Unknown' }      396     239    1.6569    -0.19947    0.021715 
    {'Employed'}      407     158    2.5759      0.2418    0.026323 
    {'Totals'  }      803     397    2.0227         NaN    0.048038 
```

```
bi = bininfo(sc,'CustIncome');
disp(bi)
```

```
           Bin            Good    Bad      Odds         WOE        InfoValue 
    _________________     ____    ___    _______    _________    __________

    {'[0,29000)'    }      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 `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). For predictors that have missing data, there is an explicit `<missing>` bin, with a corresponding WOE value computed from the data. When using `fitmodel`, the corresponding WOE value for the `<missing>` bin is applied when performing the WOE transformation.

```
[sc,mdl] = 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 = 1442.8477, Chi2Stat = 4.4974731, PValue = 0.033944979
6. Adding ResStatus, Deviance = 1438.9783, Chi2Stat = 3.86941, PValue = 0.049173805
7. Adding OtherCC, Deviance = 1434.9751, Chi2Stat = 4.0031966, PValue = 0.045414057

Generalized linear regression model:
    logit(status) ~ 1 + CustAge + ResStatus + EmpStatus + CustIncome + TmWBank + OtherCC + AMBala
    Distribution = Binomial

Estimated Coefficients:
                   Estimate       SE       tStat       pValue  
                   ________    ________    ______    __________

    (Intercept)    0.70229     0.063959     10.98    4.7498e-28
    CustAge        0.57421      0.25708    2.2335      0.025513
    ResStatus       1.3629      0.66952    2.0356       0.04179
    EmpStatus      0.88373       0.2929    3.0172      0.002551
    CustIncome     0.73535       0.2159     3.406    0.00065929
    TmWBank         1.1065      0.23267    4.7556    1.9783e-06
    OtherCC         1.0648      0.52826    2.0156      0.043841
    AMBalance       1.0446      0.32197    3.2443     0.0011775


1200 observations, 1192 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 88.5, p-value = 2.55e-16
```

Scale the scorecard points by the "points, odds, and points to double the odds (PDO)" method using the `'PointsOddsAndPDO'` argument of `formatpoints`. 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).

Display the scorecard showing the scaled points for predictors retained in the fitting model.

```
sc = formatpoints(sc,'PointsOddsAndPDO',[500 2 50]);
PointsInfo = displaypoints(sc)
```

```
PointsInfo=38×3 table
     Predictors            Bin            Points
    ____________    ______________    ______

    {'CustAge'  }    {'[0,33)'    }    54.062
    {'CustAge'  }    {'[33,37)'   }    56.282
    {'CustAge'  }    {'[37,40)'   }    60.012
    {'CustAge'  }    {'[40,46)'   }    69.636
    {'CustAge'  }    {'[46,48)'   }    77.912
    {'CustAge'  }    {'[48,51)'   }     78.86
    {'CustAge'  }    {'[51,58)'   }     80.83
    {'CustAge'  }    {'[58,Inf]'  }     96.76
    {'CustAge'  }    {'<missing>' }    64.984
    {'ResStatus'}    {'Tenant'    }    62.138
    {'ResStatus'}    {'Home Owner'}    73.248
    {'ResStatus'}    {'Other'     }    90.828
    {'ResStatus'}    {'<missing>' }    74.125
    {'EmpStatus'}    {'Unknown'   }    58.807
    {'EmpStatus'}    {'Employed'  }    86.937
    {'EmpStatus'}    {'<missing>' }       NaN
       ⋮
```

Notice that points for the `<missing>` bin for `CustAge` and `ResStatus` are explicitly shown (as `64.9836` and `74.1250`, respectively). These points are computed from the WOE value for the `<missing>` bin, and the logistic model coefficients.

For predictors that have no missing data in the training set, there is no explicit `<missing>` bin. By default the points are set to `NaN` for missing data and they lead to a score of `NaN` when running `score`. For predictors that have no explicit `<missing>` bin, use the name-value argument `'Missing'` in `formatpoints` to indicate how missing data should be treated for scoring purposes.

For the purpose of illustration, take a few rows from the original data as test data and introduce some missing data. Also introduce some invalid, or out-of-range values. For numeric data, values below the minimum (or above the maximum) allowed are considered invalid, such as a negative value for age (recall `'MinValue'` was earlier set to `0` for `CustAge` and `CustIncome`). For categorical data, invalid values are categories not explicitly included in the scorecard, for example, a residential status not previously mapped to scorecard categories, such as "House", or a meaningless string such as "abc123".

```
tdata = dataMissing(11:18,mdl.PredictorNames); % Keep only the predictors retained in the model
% Set some missing values
tdata.CustAge(1) = NaN;
tdata.ResStatus(2) = '<undefined>';
tdata.EmpStatus(3) = '<undefined>';
tdata.CustIncome(4) = NaN;
% Set some invalid values
tdata.CustAge(5) = -100;
tdata.ResStatus(6) = 'House';
tdata.EmpStatus(7) = 'Freelancer';
tdata.CustIncome(8) = -1;
disp(tdata)
```

```
    CustAge     ResStatus      EmpStatus      CustIncome    TmWBank    OtherCC    AMBalance
    _______    ___________    ___________    __________    _______    _______    _________

      NaN      Tenant         Unknown          34000         44         Yes        119.8  
       48      <undefined>    Unknown          44000         14         Yes       403.62  
       65      Home Owner     <undefined>      48000          6         No        111.88  
       44      Other          Unknown            NaN         35         No        436.41  
     -100      Other          Employed         46000         16         Yes       162.21  
       33      House          Employed         36000         36         Yes       845.02  
    39      Tenant         Freelancer       34000          40          Yes        756.26
    24      Home Owner     Employed            -1          19          Yes        449.61
```

Score the new data and see how points are assigned for missing `CustAge` and `ResStatus`, because we have an explicit bin with points for `<missing>`. However, for `EmpStatus` and `CustIncome` the `score` function sets the points to `NaN`.

```
[Scores,Points] = score(sc,tdata);
disp(Scores)
```

```
  481.2231
  520.8353
       NaN
       NaN
  551.7922
  487.9588
       NaN
       NaN
```

```
disp(Points)
```

```
    CustAge     ResStatus     EmpStatus     CustIncome     TmWBank     OtherCC     AMBalance
    _______     _________     _________     __________     _______     _______     _________

    64.984       62.138        58.807         67.893       61.858      75.622       89.922
     78.86       74.125        58.807         82.439       61.061      75.622       89.922
     96.76       73.248           NaN         96.969       51.132      50.914       89.922
    69.636       90.828        58.807            NaN       61.858      50.914       89.922
    64.984       90.828        86.937         82.439       61.061      75.622       89.922
    56.282       74.125        86.937         70.107       61.858      75.622       63.028
    60.012       62.138           NaN         67.893       61.858      75.622       63.028
    54.062       73.248        86.937            NaN       61.061      75.622       89.922
```

Use the name-value argument `'Missing'` in `formatpoints` to choose how to assign points to missing values for predictors that do not have an explicit `<missing>` bin. In this example, use the `'MinPoints'` option for the `'Missing'` argument. The minimum points for `EmpStatus` in the scorecard displayed above are `58.8072`, and for `CustIncome` the minimum points are `29.3753`.

```
sc = formatpoints(sc,'Missing','MinPoints');
[Scores,Points] = score(sc,tdata);
disp(Scores)
```

```
  481.2231
  520.8353
  517.7532
  451.3405
  551.7922
  487.9588
  449.3577
  470.2267
```

```
disp(Points)
```

```
    CustAge     ResStatus     EmpStatus     CustIncome     TmWBank     OtherCC     AMBalance
    _______     _________     _________     __________     _______     _______     _________

    64.984       62.138        58.807         67.893       61.858      75.622       89.922
     78.86       74.125        58.807         82.439       61.061      75.622       89.922
     96.76       73.248        58.807         96.969       51.132      50.914       89.922
    69.636       90.828        58.807         29.375       61.858      50.914        89.922
    64.984       90.828        86.937         82.439       61.061      75.622        89.922
    56.282       74.125        86.937         70.107       61.858      75.622        63.028
    60.012       62.138        58.807         67.893       61.858      75.622        63.028
    54.062       73.248        86.937         29.375       61.061      75.622        89.922
```

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 pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

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 the comma-separated pair consisting of 'BasePoints' and 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 the comma-separated pair consisting of 'Missing' and a character vector with a value for NoScore, ZeroWOE, 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.

Note When using the `creditscorecard` name-value argument `'BinMissingData'` with a value of `true`, missing data for numeric and categorical predictors is binned in a separate bin labeled `<missing>`. The `<missing>` bin only contains missing values for a predictor and does not contain invalid or out-of-range values for a predictor.

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 the comma-separated pair consisting of `'Round'` and 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).

For more information and an example of using the `'Round'` name-value pair argument, see “Rounding and Default Probabilities” on page 15-1674.

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 as the comma-separated pair consisting of `'ShiftAndSlope'` and a 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 as the comma-separated pair consisting of `'PointsOddsAndPDO'` and a 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 the comma-separated pair consisting of 'WorstAndBestScores' and 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.

Version History

Introduced in R2014b

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

`creditscorecard` | `autobinning` | `bininfo` | `predictorinfo` | `modifypredictor` | `plotbins` | `modifybins` | `bindata` | `fitmodel` | `displaypoints` | `score` | `setmodel` | `probdefault` | `validatemodel`

Topics

“Case Study for Credit Scorecard Analysis” on page 8-70
“Credit Scorecard Modeling with Missing Values” on page 8-56
“Troubleshooting Credit Scorecard Results” on page 8-63
“Credit Scorecard Modeling Workflow” on page 8-51
“About Credit Scorecards” on page 8-47

displaypoints

Return points per predictor per bin

Syntax

```
PointsInfo = displaypoints(sc)
[PointsInfo,MinScore,MaxScore] = displaypoints(sc)
[PointsInfo,MinScore,MaxScore] = displaypoints( ___ ,Name,Value)
```

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.

`[PointsInfo,MinScore,MaxScore] = displaypoints( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

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-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 + AMBala
    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=37×3 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
    {'CustAge'   }    {'<missing>'   }          NaN
    {'ResStatus' }    {'Tenant'      }    -0.031252
    {'ResStatus' }    {'Home Owner'  }      0.12696
    {'ResStatus' }    {'Other'       }      0.37641
    {'ResStatus' }    {'<missing>'   }          NaN
    {'EmpStatus' }    {'Unknown'     }    -0.076317
    {'EmpStatus' }    {'Employed'    }      0.31449
    {'EmpStatus' }    {'<missing>'   }          NaN
    {'CustIncome'}    {'[-Inf,29000)'}     -0.45716
      ⋮
```

`displaypoints` always displays a `'<missing>'` bin for each predictor. The value of the `'<missing>'` bin comes from the initial `creditscorecard` object, and the `'<missing>'` bin is set to `NaN` whenever the scorecard model has no information on how to assign points to missing data.

To configure the points for the `'<missing>'` bin, you must use the initial `creditscorecard` object. For predictors that have missing values in the training set, the points for the `'<missing>'` bin are estimated from the data if the `'BinMissingData'` name-value pair argument is set to `true` using `creditscorecard`. When the `'BinMissingData'` parameter is set to `false`, or when the data contains no missing values in the training set, use the `'Missing'` name-value pair argument in `formatpoints` to indicate how to assign points to the missing data.

Display Unscaled Points When Using Missing Data

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` with missing values.

```
load CreditCardData.mat
head(dataMissing,5)
```

```
    CustID    CustAge    TmAtAddress     ResStatus     EmpStatus    CustIncome    TmWBank    Othe
    ______    _______    ___________    ___________    _________    __________    _______    ____

      1          53          62         <undefined>    Unknown        50000         55         Ye
      2          61          22         Home Owner     Employed       52000         25         Ye
      3          47          30         Tenant         Employed       37000         61         No
      4         NaN          75         Home Owner     Employed       53000         20         Ye
      5          68          56         Home Owner     Employed       53000         14         Ye
```

```
fprintf('Number of rows: %d\n',height(dataMissing))
```

```
Number of rows: 1200
```

```
fprintf('Number of missing values CustAge: %d\n',sum(ismissing(dataMissing.CustAge)))
```

```
Number of missing values CustAge: 30
```

```
fprintf('Number of missing values ResStatus: %d\n',sum(ismissing(dataMissing.ResStatus)))
```

```
Number of missing values ResStatus: 40
```

Use `creditscorecard` with the name-value argument `'BinMissingData'` set to `true` to bin the missing numeric or categorical data in a separate bin. Apply automatic binning.

```
sc = creditscorecard(dataMissing,'IDVar','CustID','BinMissingData',true);
sc = autobinning(sc);

disp(sc)
```

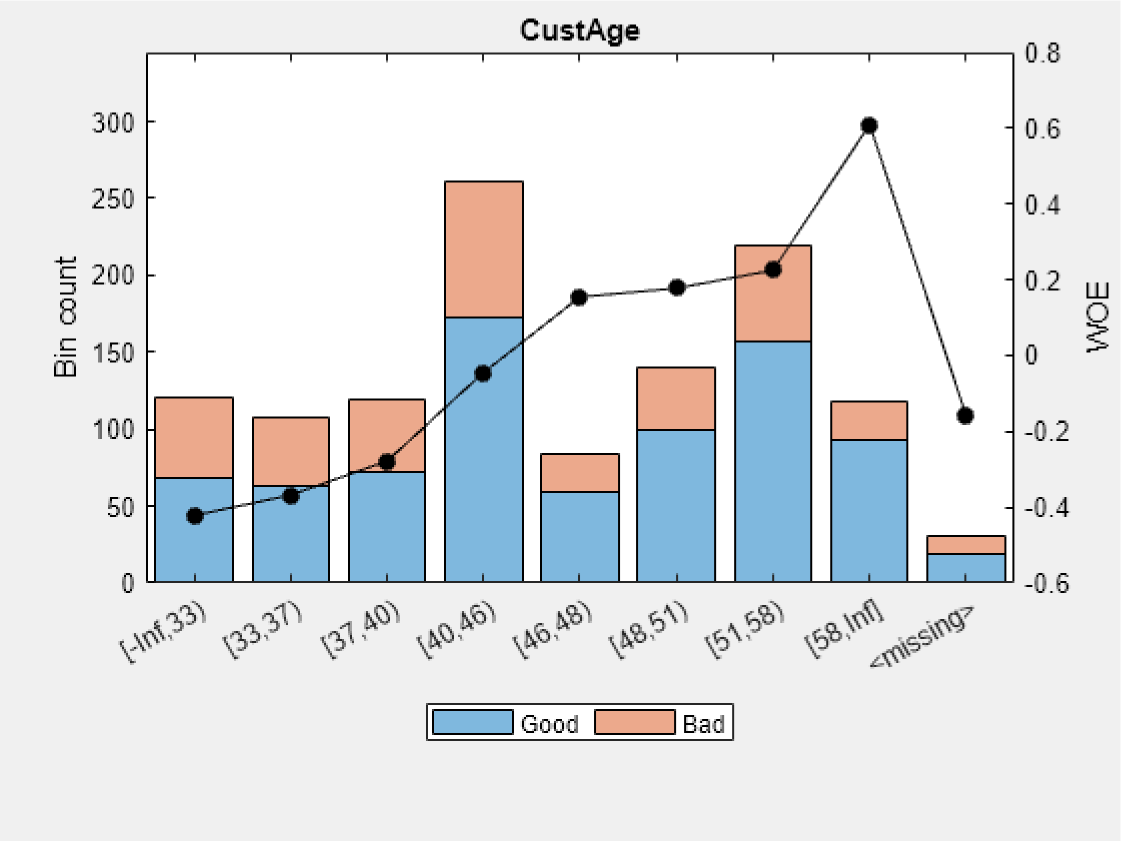
```
  creditscorecard with properties:

                GoodLabel: 0
              ResponseVar: 'status'
               WeightsVar: ''
                 VarNames: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustI
        NumericPredictors: {'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBalance'  'Uti
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
           BinMissingData: 1
                    IDVar: 'CustID'
            PredictorVars: {'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIncome'  'Tr
                     Data: [1200x11 table]
```

Display and plot bin information for numeric data for `'CustAge'` that includes missing data in a separate bin labelled `<missing>`.

```
[bi,cp] = bininfo(sc,'CustAge');
disp(bi)
```

```
         Bin           Good    Bad     Odds        WOE       InfoValue 
    _______________    ____    ___    ______    ________    __________

    {'[-Inf,33)'}       69     52     1.3269    -0.42156      0.018993
    {'[33,37)'  }       63     45        1.4    -0.36795      0.012839
    {'[37,40)'  }       72     47     1.5319     -0.2779     0.0079824
    {'[40,46)'  }      172     89     1.9326    -0.04556     0.0004549
    {'[46,48)'  }       59     25       2.36     0.15424     0.0016199
    {'[48,51)'  }       99     41     2.4146     0.17713     0.0035449
    {'[51,58)'  }      157     62     2.5323     0.22469     0.0088407
    {'[58,Inf]' }       93     25       3.72     0.60931      0.032198
    {'<missing>'}       19     11     1.7273    -0.15787    0.00063885
    {'Totals'   }      803    397     2.0227         NaN      0.087112
```

```
plotbins(sc,'CustAge')
```

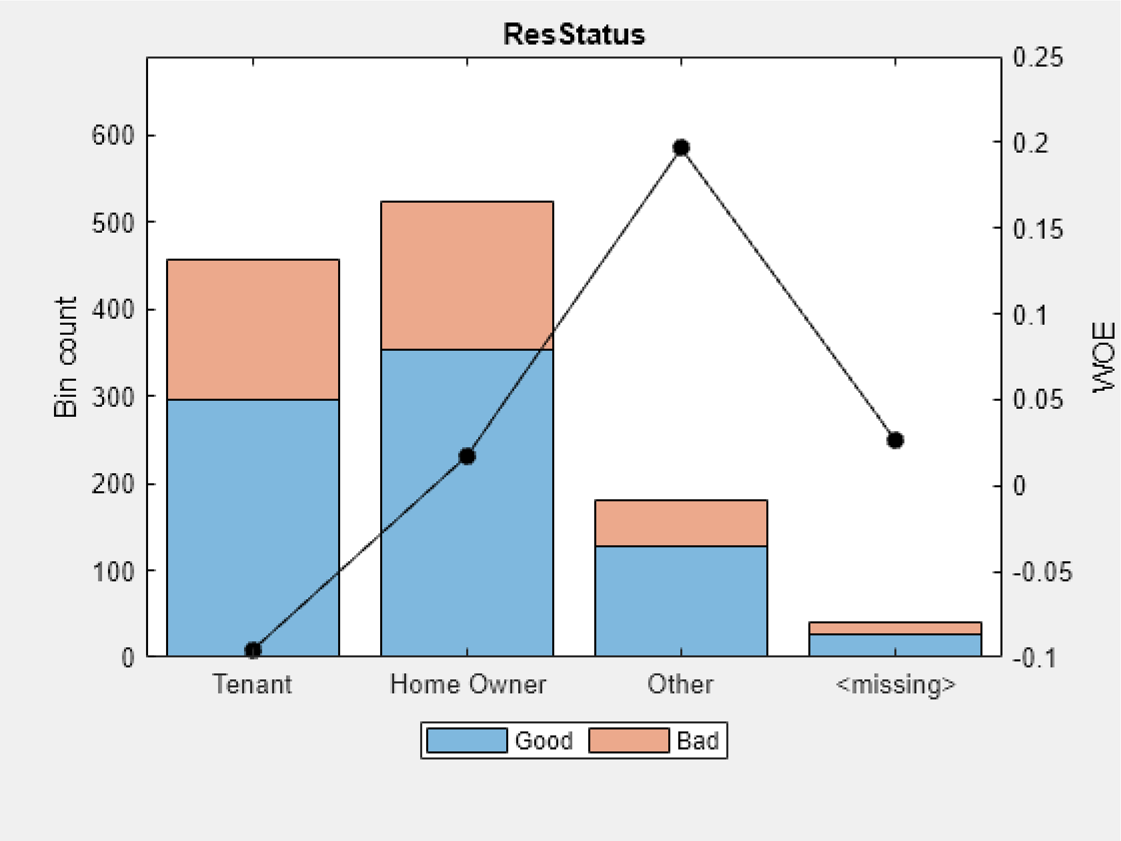


Display and plot bin information for categorical data for `'ResStatus'` that includes missing data in a separate bin labelled `<missing>`.

```
[bi,cg] = bininfo(sc,'ResStatus');
disp(bi)
```

```
         Bin           Good    Bad     Odds        WOE       InfoValue 
    ______________     ____    ___    ______    _________    __________

    {'Tenant'     }    296     161    1.8385    -0.095463     0.0035249
    {'Home Owner'}     352     171    2.0585     0.017549    0.00013382
    {'Other'      }    128      52    2.4615      0.19637     0.0055808
    {'<missing>' }      27      13    2.0769     0.026469    2.3248e-05
    {'Totals'     }    803     397    2.0227          NaN     0.0092627
```

```
plotbins(sc,'ResStatus')
```



For the `'CustAge'` and `'ResStatus'` predictors, there is missing data (`NaN`s and `<undefined>`) in the training data, and the binning process estimates a WOE value of `-0.15787` and `0.026469` respectively for missing data in these predictors, as shown above.

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). For predictors that have missing data, there is an explicit `<missing>` bin, with a corresponding WOE value computed from the data. When using `fitmodel`, the corresponding WOE value for the <missing> bin is applied when performing the WOE transformation.

```
[sc,mdl] = 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 = 1442.8477, Chi2Stat = 4.4974731, PValue = 0.033944979
6. Adding ResStatus, Deviance = 1438.9783, Chi2Stat = 3.86941, PValue = 0.049173805
7. Adding OtherCC, Deviance = 1434.9751, Chi2Stat = 4.0031966, PValue = 0.045414057

Generalized linear regression model:
    logit(status) ~ 1 + CustAge + ResStatus + EmpStatus + CustIncome + TmWBank + OtherCC + AMBala
    Distribution = Binomial

Estimated Coefficients:
                   Estimate       SE       tStat       pValue   
                   ________    ________    ______    __________

    (Intercept)    0.70229     0.063959     10.98    4.7498e-28
    CustAge        0.57421      0.25708    2.2335      0.025513
    ResStatus       1.3629      0.66952    2.0356       0.04179
    EmpStatus      0.88373       0.2929    3.0172      0.002551
    CustIncome     0.73535       0.2159     3.406    0.00065929
    TmWBank         1.1065      0.23267    4.7556    1.9783e-06
    OtherCC         1.0648      0.52826    2.0156      0.043841
    AMBalance       1.0446      0.32197    3.2443     0.0011775


1200 observations, 1192 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 88.5, p-value = 2.55e-16
```

Display unscaled points for predictors retained in the fitting model (to scale points use `formatpoints`).

```
PointsInfo = displaypoints(sc)
```

```
PointsInfo=38×3 table
     Predictors          Bin           Points  
    _____________    ______________    _________

    {'CustAge'  }    {'[-Inf,33)' }     -0.14173
    {'CustAge'  }    {'[33,37)'   }     -0.11095
    {'CustAge'  }    {'[37,40)'   }    -0.059244
    {'CustAge'  }    {'[40,46)'   }     0.074167
    {'CustAge'  }    {'[46,48)'   }       0.1889
    {'CustAge'  }    {'[48,51)'   }      0.20204
    {'CustAge'  }    {'[51,58)'   }      0.22935
    {'CustAge'  }    {'[58,Inf]'  }      0.45019
    {'CustAge'  }    {'<missing>' }    0.0096749
    {'ResStatus'}    {'Tenant'    }    -0.029778
    {'ResStatus'}    {'Home Owner'}      0.12425
    {'ResStatus'}    {'Other'     }      0.36796
    {'ResStatus'}    {'<missing>' }       0.1364
    {'EmpStatus'}    {'Unknown'   }    -0.075948
    {'EmpStatus'}    {'Employed'  }      0.31401
    {'EmpStatus'}    {'<missing>' }          NaN
      ⋮
```

Notice that points for the <missing> bin for CustAge and ResStatus are explicitly shown. These points are computed from the WOE value for the <missing> bin and the logistic model coefficients.

For predictors that have no missing data in the training set, there is no explicit <missing> bin, and by default the points are set to NaN for missing data, and they lead to a score of NaN when running score. For predictors that have no explicit <missing> bin, use the name-value argument 'Missing' in formatpoints to indicate how missing data should be treated for scoring purposes.

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-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 + AMBala
    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.

```
[PointsInfo,MinScore,MaxScore] = displaypoints(sc)
```

```
PointsInfo=37×3 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
    {'CustAge'   }    {'<missing>'   }       NaN
    {'ResStatus' }    {'Tenant'      }    62.421
    {'ResStatus' }    {'Home Owner'  }    82.276
    {'ResStatus' }    {'Other'       }    113.58
    {'ResStatus' }    {'<missing>'   }       NaN
    {'EmpStatus' }    {'Unknown'     }    56.765
    {'EmpStatus' }    {'Employed'    }    105.81
    {'EmpStatus' }    {'<missing>'   }       NaN
    {'CustIncome'}    {'[-Inf,29000)'}    8.9706
       ⋮

MinScore = 300.0000

MaxScore = 850
```

Notice that, as expected, the values of `MinScore` and `MaxScore` correspond to the worst and best possible scores.

Separate the Base Points From the Total Points

This example shows how to use `displaypoints` after a model is fitted to separate the base points from the rest of the points assigned to each predictor variable. The name-value pair argument `'BasePoints'` in the `formatpoints` function is a boolean that serves this purpose. By default, the base points are spread across all variables in the scorecard.

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011). Use the `'IDVar'` argument in the `creditscorecard` function to indicate that `'CustID'` contains ID information and should not be included as a predictor variable.

```
load CreditCardData
sc = creditscorecard(data,'IDVar','CustID');
```

Perform automatic binning to bin for all predictors.

```
sc = autobinning(sc);
```

Fit a linear regression model using default parameters.

```
sc = fitmodel(sc);
```

```
1. Adding CustIncome, Deviance = 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 + AMBala
    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=38×3 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
    {'CustAge'   }    {'<missing>' }          NaN
    {'ResStatus' }    {'Tenant'    }     -0.13159
    {'ResStatus' }    {'Home Owner'}     0.026616
    {'ResStatus' }    {'Other'     }      0.27607
    {'ResStatus' }    {'<missing>' }          NaN
    {'EmpStatus' }    {'Unknown'   }     -0.17666
    {'EmpStatus' }    {'Employed'  }      0.21415
    {'EmpStatus' }    {'<missing>' }          NaN
      ⋮
```

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-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 + AMBala
    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=37×3 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
    {'CustAge'   }    {'<missing>'   }          NaN
    {'ResStatus' }    {'Tenant'      }    -0.031252
    {'ResStatus' }    {'Home Owner'  }      0.12696
    {'ResStatus' }    {'Other'       }      0.37641
    {'ResStatus' }    {'<missing>'   }          NaN
    {'EmpStatus' }    {'Unknown'     }    -0.076317
    {'EmpStatus' }    {'Employed'    }      0.31449
    {'EmpStatus' }    {'<missing>'   }          NaN
    {'CustIncome'}    {'[-Inf,29000)'}     -0.45716
      ⋮


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 58'};
sc = modifybins(sc,'CustAge','BinLabels',labels);
```

Rerun `displaypoints` to verify the updated bin labels.

```
[PointsInfo,MinScore,MaxScore] = displaypoints(sc)
```

```
PointsInfo=37×3 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
    {'CustAge'   }    {'<missing>'   }          NaN
    {'ResStatus' }    {'Tenant'      }    -0.031252
    {'ResStatus' }    {'Home Owner'  }      0.12696
    {'ResStatus' }    {'Other'       }      0.37641
    {'ResStatus' }    {'<missing>'   }          NaN
    {'EmpStatus' }    {'Unknown'     }    -0.076317
    {'EmpStatus' }    {'Employed'    }      0.31449
    {'EmpStatus' }    {'<missing>'   }          NaN
    {'CustIncome'}    {'[-Inf,29000)'}     -0.45716
      ⋮

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-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 + AMBala
    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
    {'CustAge'  }    {'<missing>' }       NaN
    {'ResStatus'}    {'Tenant'    }    62.031
    {'ResStatus'}    {'Home Owner'}    73.444
```

```
 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.

Display Points for `creditscorecard` Object That Contains Missing Data

To create a `creditscorecard` object using the `CreditCardData.mat` file, load the `data` (using a dataset from Refaat 2011). Using the `dataMissing` dataset, set the `'BinMissingData'` indicator to `true`.

```
load CreditCardData.mat
sc = creditscorecard(dataMissing,'BinMissingData',true);
```

Use `autobinning` with the `creditscorecard` object.

```
sc = autobinning(sc);
```

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=5×6 table
         Bin           Good    Bad      Odds        WOE        InfoValue 
    ______________     ____    ___     ______    _________    __________

    {'Tenant'    }     296     161     1.8385    -0.095463     0.0035249
    {'Home Owner'}     352     171     2.0585     0.017549    0.00013382
    {'Other'     }     128      52     2.4615      0.19637     0.0055808
    {'<missing>' }      27      13     2.0769     0.026469    2.3248e-05
    {'Totals'    }     803     397     2.0227          NaN     0.0092627

cg=3×2 table
       Category        BinNumber
    ______________     _________

    {'Tenant'    }         1    
    {'Home Owner'}         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 `creditscorecard` object.

```
cg.BinNumber(3) = 2;
sc = modifybins(sc,'ResStatus','Catg',cg);
```

Display the updated bin information using `bininfo`. 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=4×6 table
         Bin         Good    Bad      Odds         WOE       InfoValue 
    _____________    ____    ___    ______    _________    __________

    {'Group1'   }    296     161    1.8385    -0.095463     0.0035249
    {'Group2'   }    480     223    2.1525     0.062196     0.0022419
    {'<missing>'}     27      13    2.0769     0.026469    2.3248e-05
    {'Totals'   }    803     397    2.0227          NaN       0.00579

cg=3×2 table
       Category         BinNumber
    ______________    _________

    {'Tenant'    }        1    
    {'Home Owner'}        2    
    {'Other'     }        2    
```

Use `formatpoints` with the `'Missing'` name-value pair argument to indicate that missing data is assigned `'maxpoints'`.

```
sc = formatpoints(sc,'BasePoints',true,'Missing','maxpoints','WorstAndBest',[300 800]);
```

Use `fitmodel` to fit the model.

```
sc = fitmodel(sc,'VariableSelection','fullmodel','Display','Off');
```

Then use `displaypoints` (Risk Management Toolbox) with the `creditscorecard` object to return a table of points for all bins of all predictor variables used in the `compactCreditScorecard` object. By setting the `displaypoints` (Risk Management Toolbox) name-value pair argument for '`ShowCategoricalMembers`' to `true`, all the members contained in each individual group are displayed.

```
[PointsInfo,MinScore,MaxScore] = displaypoints(sc,'ShowCategoricalMembers',true)
```

```
PointsInfo=51×3 table
      Predictors             Bin           Points 
    _______________    ______________    _______

    {'BasePoints' }    {'BasePoints'}     535.25
    {'CustID'     }    {'[-Inf,121)'}     12.085
    {'CustID'     }    {'[121,241)' }     5.4738
    {'CustID'     }    {'[241,1081)'}    -1.4061
    {'CustID'     }    {'[1081,Inf]'}    -7.2217
    {'CustID'     }    {'<missing>' }     12.085
    {'CustAge'    }    {'[-Inf,33)' }    -25.973
    {'CustAge'    }    {'[33,37)'   }     -22.67
    {'CustAge'    }    {'[37,40)'   }    -17.122
    {'CustAge'    }    {'[40,46)'   }    -2.8071
    {'CustAge'    }    {'[46,48)'   }     9.5034
    {'CustAge'    }    {'[48,51)'   }     10.913
    {'CustAge'    }    {'[51,58)'   }     13.844
    {'CustAge'    }    {'[58,Inf]'  }     37.541
    {'CustAge'    }    {'<missing>' }    -9.7271
    {'TmAtAddress'}    {'[-Inf,23)' }    -9.3683
         ⋮


MinScore = 300.0000

MaxScore = 800.0000
```

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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[PointsInfo,MinScore,MaxScore] = displaypoints(sc, ‘ShowCategoricalMembers’,true)`

`ShowCategoricalMembers` — Indicator for how to display bins labels of categories that were grouped together

`false` (default) | `true` or `false`

Indicator for how to display bins labels of categories that were grouped together, specified as the comma-separated pair consisting of `'ShowCategoricalMembers'` and a logical scalar with a value of `true` or `false`.

By default, when `'ShowCategoricalMembers'` is `false`, bin labels are displayed as `Group1`, `Group2`,...,`Group`*n*, or if the bin labels were modified in `creditscorecard`, then the user-defined bin label names are displayed.

If `'ShowCategoricalMembers'` is `true`, all the members contained in each individual group are displayed.

Data Types: `logical`

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_1	Bin_12	Points_12
Predictor_1	Bin_13	Points_13
	...	...
Predictor_1	`'<missing>'`	`NaN` (Default)
Predictor_2	Bin_21	Points_21
Predictor_2	Bin_22	Points_22
Predictor_2	Bin_23	Points_23
	...	...
Predictor_2	`'<missing>'`	`NaN` (Default)
Predictor_*j*	Bin_*ji*	Points_*ji*
	...	...
Predictor_*j*	`'<missing>'`	`NaN` (Default)

`displaypoints` always displays a `'<missing>'` bin for each predictor. The value of the `'<missing>'` bin comes from the initial `creditscorecard` object, and the `'<missing>'` bin is set to `NaN` whenever the scorecard model has no information on how to assign points to missing data.

To configure the points for the `'<missing>'` bin, you must use the initial `creditscorecard` object. For predictors that have missing values in the training set, the points for the `'<missing>'` bin are estimated from the data if the `'BinMissingData'` name-value pair argument for is set to `true` using `creditscorecard`. When the `'BinMissingData'` parameter is set to `false`, or when the data contains no missing values in the training set, use the `'Missing'` name-value pair argument in `formatpoints` to indicate how to assign points to the missing data.

Another option is to use `fillmissing` to specify replacement "fill" values for predictors with a `NaN` or `<undefined>` value. If you use `fillmissing`, then the `displaypoints` `'<missing>'` row has the same points as the bin associated with the fill value.

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 WOE*j*(*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.

Version History

Introduced in R2014b

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

`creditscorecard` | `autobinning` | `bininfo` | `predictorinfo` | `modifypredictor` | `plotbins` | `fillmissing` | `modifybins` | `bindata` | `fitmodel` | `formatpoints` | `score` | `setmodel` | `probdefault` | `validatemodel`

Topics

“Case Study for Credit Scorecard Analysis” on page 8-70
“Credit Scorecard Modeling with Missing Values” on page 8-56
“Troubleshooting Credit Scorecard Results” on page 8-63
“Credit Scorecard Modeling Workflow” on page 8-51

"About Credit Scorecards" on page 8-47

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: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustI
        NumericPredictors: {'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBalance'  'Uti
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
           BinMissingData: 0
                    IDVar: 'CustID'
            PredictorVars: {'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIncome'  'T
                     Data: [1200x11 table]
```

Perform automatic binning.

```
sc = autobinning(sc)
```

```
sc = 
  creditscorecard with properties:

                GoodLabel: 0
              ResponseVar: 'status'
               WeightsVar: ''
                 VarNames: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustI
        NumericPredictors: {'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBalance'  'Uti
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
           BinMissingData: 0
                    IDVar: 'CustID'
            PredictorVars: {'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIncome'  'T
                     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-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 + AMBala
    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: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustI
      NumericPredictors: {'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBalance'  'Uti
  CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
         BinMissingData: 0
                  IDVar: 'CustID'
          PredictorVars: {'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIncome'  'T
                   Data: [1200x12 table]
```

Perform automatic binning.

```
sc = autobinning(sc)
```

```
sc = 
  creditscorecard with properties:

              GoodLabel: 0
            ResponseVar: 'status'
             WeightsVar: 'RowWeights'
               VarNames: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustI
      NumericPredictors: {'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBalance'  'Uti
  CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
         BinMissingData: 0
```

```
                   IDVar: 'CustID'
           PredictorVars: {'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIncome'  'Tr
                    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: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustI
        NumericPredictors: {'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBalance'  'Uti
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
           BinMissingData: 0
                    IDVar: 'CustID'
```

```
            PredictorVars: {'CustAge'    'TmAtAddress'    'ResStatus'    'EmpStatus'    'CustIncome'    'Tr
                     Data: [1200x11 table]
```

Perform automatic binning.

```
sc = autobinning(sc,'Algorithm','EqualFrequency')
```

```
sc = 
  creditscorecard with properties:

                GoodLabel: 0
              ResponseVar: 'status'
               WeightsVar: ''
                 VarNames: {'CustID'    'CustAge'    'TmAtAddress'    'ResStatus'    'EmpStatus'    'CustI
        NumericPredictors: {'CustAge'    'TmAtAddress'    'CustIncome'    'TmWBank'    'AMBalance'    'Uti
    CategoricalPredictors: {'ResStatus'    'EmpStatus'    'OtherCC'}
           BinMissingData: 0
                    IDVar: 'CustID'
            PredictorVars: {'CustAge'    'TmAtAddress'    'ResStatus'    'EmpStatus'    'CustIncome'    'Tr
                     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:
    logit(status) ~ 1 + CustAge + TmAtAddress + ResStatus + EmpStatus + CustIncome + TmWBank + O
    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
```

Fit a Stepwise Logistic Model When Using Missing Data

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `dataMissing` with missing values.

```
load CreditCardData.mat
head(dataMissing,5)
```

```
    CustID    CustAge    TmAtAddress     ResStatus     EmpStatus    CustIncome    TmWBank    Othe
    ______    _______    ___________    ___________    _________    __________    _______    ___

      1          53          62         <undefined>    Unknown        50000         55         Ye
      2          61          22         Home Owner     Employed       52000         25         Ye
      3          47          30         Tenant         Employed       37000         61         No
      4         NaN          75         Home Owner     Employed       53000         20         Ye
      5          68          56         Home Owner     Employed       53000         14         Ye
```

```
fprintf('Number of rows: %d\n',height(dataMissing))
```

```
Number of rows: 1200
```

```
fprintf('Number of missing values CustAge: %d\n',sum(ismissing(dataMissing.CustAge)))
```

```
Number of missing values CustAge: 30
```

```
fprintf('Number of missing values ResStatus: %d\n',sum(ismissing(dataMissing.ResStatus)))
```

```
Number of missing values ResStatus: 40
```

Use `creditscorecard` with the name-value argument `'BinMissingData'` set to `true` to bin the missing numeric or categorical data in a separate bin.

```
sc = creditscorecard(dataMissing,'IDVar','CustID','BinMissingData',true);
sc = autobinning(sc);
disp(sc)
```

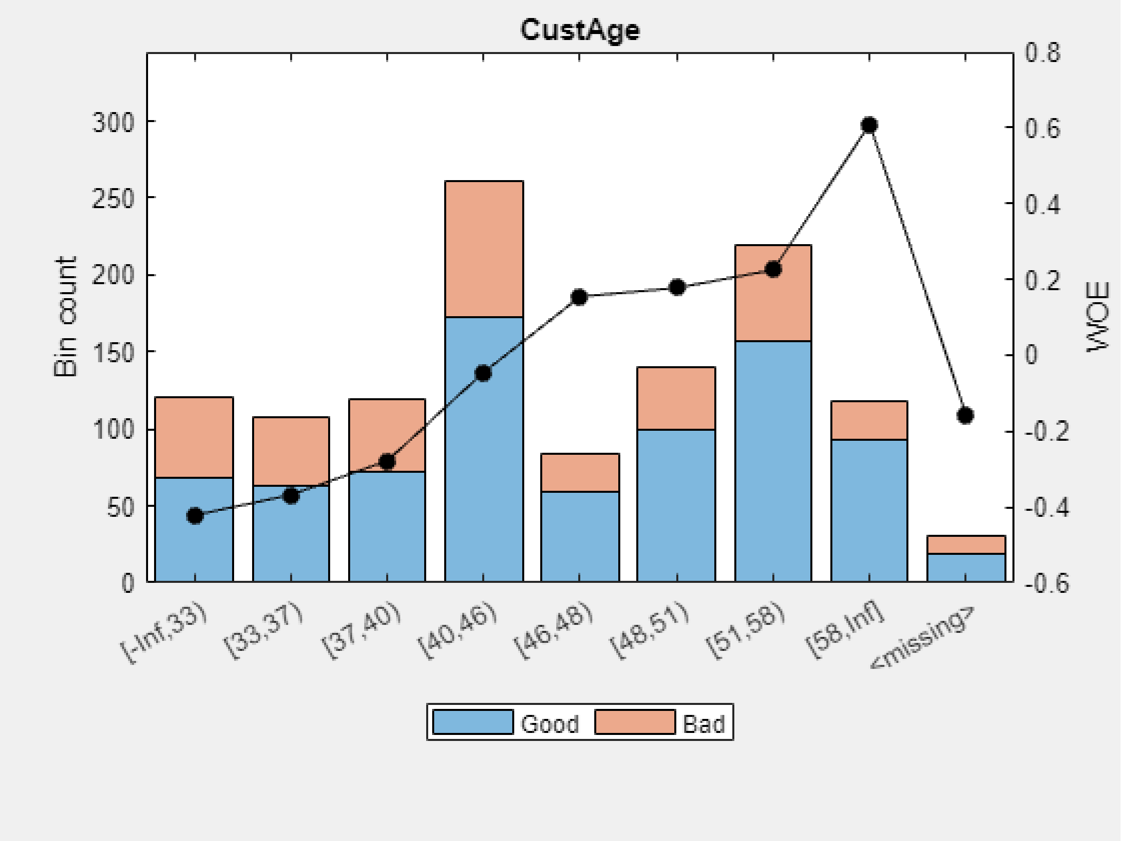
```
  creditscorecard with properties:

                GoodLabel: 0
              ResponseVar: 'status'
               WeightsVar: ''
                 VarNames: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustI
        NumericPredictors: {'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBalance'  'Uti
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
           BinMissingData: 1
                    IDVar: 'CustID'
            PredictorVars: {'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIncome'  'T
                     Data: [1200×11 table]
```

Display and plot bin information for numeric data for `'CustAge'` that includes missing data in a separate bin labelled `<missing>`.

```
[bi,cp] = bininfo(sc,'CustAge');
disp(bi)
```

```
         Bin         Good    Bad     Odds       WOE       InfoValue
    _____________    ____    ___    ______    ________    __________

    {'[-Inf,33)'}     69      52    1.3269    -0.42156      0.018993
    {'[33,37)'  }     63      45       1.4    -0.36795      0.012839
    {'[37,40)'  }     72      47    1.5319     -0.2779     0.0079824
    {'[40,46)'  }    172      89    1.9326    -0.04556     0.0004549
    {'[46,48)'  }     59      25      2.36     0.15424     0.0016199
    {'[48,51)'  }     99      41    2.4146     0.17713     0.0035449
    {'[51,58)'  }    157      62    2.5323     0.22469     0.0088407
    {'[58,Inf]' }     93      25      3.72     0.60931      0.032198
    {'<missing>'}     19      11    1.7273    -0.15787    0.00063885
    {'Totals'   }    803     397    2.0227         NaN      0.087112
```

```
plotbins(sc,'CustAge')
```

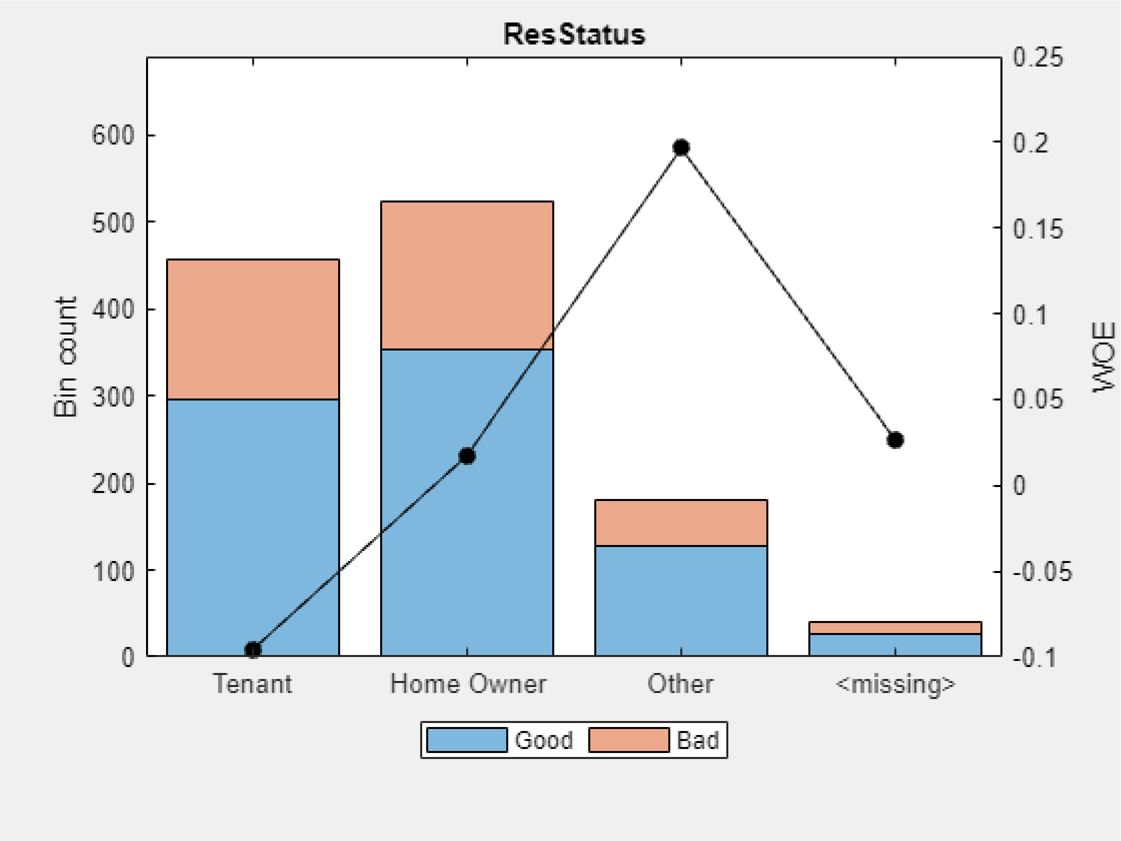


Display and plot bin information for categorical data for 'ResStatus' that includes missing data in a separate bin labelled <missing>.

```
[bi,cg] = bininfo(sc,'ResStatus');
disp(bi)
```

```
         Bin           Good    Bad     Odds        WOE        InfoValue 
    ______________     ____    ___    ______    _________    __________

    {'Tenant'     }    296     161    1.8385    -0.095463     0.0035249
    {'Home Owner'}     352     171    2.0585     0.017549    0.00013382
    {'Other'      }    128      52    2.4615      0.19637     0.0055808
    {'<missing>' }      27      13    2.0769     0.026469    2.3248e-05
    {'Totals'     }    803     397    2.0227          NaN     0.0092627
```

```
plotbins(sc,'ResStatus')
```



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). For predictors that have missing data, there is an explicit `<missing>` bin, with a corresponding WOE value computed from the data. When using `fitmodel`, the corresponding WOE value for the <missing> bin is applied when performing the WOE transformation. For example, a missing value for customer age (`CustAge`) is replaced with `-0.15787` which is the WOE value for the `<missing>` bin for the `CustAge` predictor. However, when `'BinMissingData'` is false, a missing value for `CustAge` remains as missing (`NaN`) when applying the WOE transformation.

```
[sc,mdl] = 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 = 1442.8477, Chi2Stat = 4.4974731, PValue = 0.033944979
6. Adding ResStatus, Deviance = 1438.9783, Chi2Stat = 3.86941, PValue = 0.049173805
7. Adding OtherCC, Deviance = 1434.9751, Chi2Stat = 4.0031966, PValue = 0.045414057

Generalized linear regression model:
    logit(status) ~ 1 + CustAge + ResStatus + EmpStatus + CustIncome + TmWBank + OtherCC + AMBala
    Distribution = Binomial

Estimated Coefficients:
                   Estimate       SE       tStat       pValue
                   ________    ________    ______    __________
    (Intercept)    0.70229     0.063959     10.98    4.7498e-28
    CustAge        0.57421      0.25708    2.2335      0.025513
    ResStatus       1.3629      0.66952    2.0356       0.04179
    EmpStatus      0.88373       0.2929    3.0172      0.002551
    CustIncome     0.73535       0.2159     3.406    0.00065929
    TmWBank         1.1065      0.23267    4.7556    1.9783e-06
    OtherCC         1.0648      0.52826    2.0156      0.043841
    AMBalance       1.0446      0.32197    3.2443     0.0011775


1200 observations, 1192 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 88.5, p-value = 2.55e-16
```

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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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 as the comma-separated pair consisting of `'PredictorVars'` and a cell array of character vectors. When provided, the `creditscorecard` object property `PredictorsVars` is updated. Note that the order of predictors in the original dataset is enforced, regardless of the order in which `'PredictorVars'` is provided. When not provided, the predictors used to create the `creditscorecard` object (by using `creditscorecard`) are used.

Data Types: `cell`

`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 the comma-separated pair consisting of `'VariableSelection'` and 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 as the comma-separated pair consisting of `'StartingModel'` and 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 as the comma-separated pair consisting of `'Display'` and 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`.

More About

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 Toolbox functionality 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 15-1807 and “Credit Scorecard Modeling Using Observation Weights” on page 8-54.

Therefore, the credit scorecard points and final score depend on the observation weights through both the logistic model coefficients and the WOE values.

Models

A logistic regression model is used in the `creditscorecard` object.

For the model, the probability of being “Bad” is defined as: `ProbBad = exp(-s) / (1 + exp(-s))`.

Version History

Introduced in R2014b

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

`fitConstrainedModel` | `creditscorecard` | `autobinning` | `bininfo` | `predictorinfo` | `modifypredictor` | `plotbins` | `modifybins` | `bindata` | `displaypoints` | `formatpoints` | `score` | `stepwiseglm` | `fitglm` | `setmodel` | `probdefault` | `validatemodel` | `GeneralizedLinearModel`

Topics

“Case Study for Credit Scorecard Analysis” on page 8-70
“Credit Scorecard Modeling with Missing Values” on page 8-56
“Troubleshooting Credit Scorecard Results” on page 8-63
“Credit Scorecard Modeling Workflow” on page 8-51
“About Credit Scorecards” on page 8-47
“Credit Scorecard Modeling Using Observation Weights” on page 8-54
“What Are Generalized Linear Models?”

fitConstrainedModel

Fit logistic regression model to Weight of Evidence (WOE) data subject to constraints on model coefficients

Syntax

```
[sc,mdl] = fitConstrainedModel(sc)
```

```
[sc,mdl] = fitConstrainedModel( ___ ,Name,Value)
```

Description

`[sc,mdl] = fitConstrainedModel(sc)` fits a logistic regression model to the Weight of Evidence (WOE) data subject to equality, inequality, or bound constraints on the model coefficients. `fitConstrainedModel` stores the model predictor names and corresponding coefficients in an updated `creditscorecard` object `sc` and returns the `GeneralizedLinearModel` object `mdl` which contains the fitted model.

`[sc,mdl] = fitConstrainedModel( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Credit Scorecards with Constrained Logistic Regression Coefficients

To compute scores for a `creditscorecard` object with constraints for equality, inequality, or bounds on the coefficients of the logistic regression model, use `fitConstrainedModel`. Unlike `fitmodel`, `fitConstrainedModel` solves for both the unconstrained and constrained problem. The current solver used to minimize an objective function for `fitConstrainedModel` is `fmincon`, from the Optimization Toolbox™.

This example has three main sections. First, `fitConstrainedModel` is used to solve for the coefficients in the unconstrained model. Then, `fitConstrainedModel` demonstrates how to use several types of constraints. Finally, `fitConstrainedModel` uses bootstrapping for the significance analysis to determine which predictors to reject from the model.

Create the `creditscorecard` Object and Bin data

```
load CreditCardData.mat
sc = creditscorecard(data,'IDVar','CustID');
sc = autobinning(sc);
```

Unconstrained Model Using `fitConstrainedModel`

Solve for the unconstrained coefficients using `fitConstrainedModel` with default values for the input parameters. `fitConstrainedModel` uses the internal optimization solver `fmincon` from the Optimization Toolbox™. If you do not set any constraints, `fmincon` treats the model as an unconstrained optimization problem. The default parameters for the `LowerBound` and `UpperBound` are `-Inf` and `+Inf`, respectively. For the equality and inequality constraints, the default is an empty numeric array.

```
[sc1,mdl1] = fitConstrainedModel(sc);
coeff1 = mdl1.Coefficients.Estimate;
disp(mdl1.Coefficients);
```

```
                   Estimate 
                   _________

    (Intercept)      0.70246
    CustAge           0.6057
    TmAtAddress       1.0381
    ResStatus         1.3794
    EmpStatus        0.89648
    CustIncome       0.70179
    TmWBank           1.1132
    OtherCC           1.0598
    AMBalance         1.0572
    UtilRate       -0.047597
```

Unlike `fitmodel` which gives *p*-values, when using `fitConstrainedModel`, you must use bootstrapping to find out which predictors are rejected from the model, when subject to constraints. This is illustrated in the "Significance Bootstrapping" section.

Using `fitmodel` to Compare the Results and Calibrate the Model

`fitmodel` fits a logistic regression model to the Weight-of-Evidence (WOE) data and there are no constraints. You can compare the results from the "Unconstrained Model Using fitConstrainedModel" section with those of `fitmodel` to verify that the model is well calibrated.

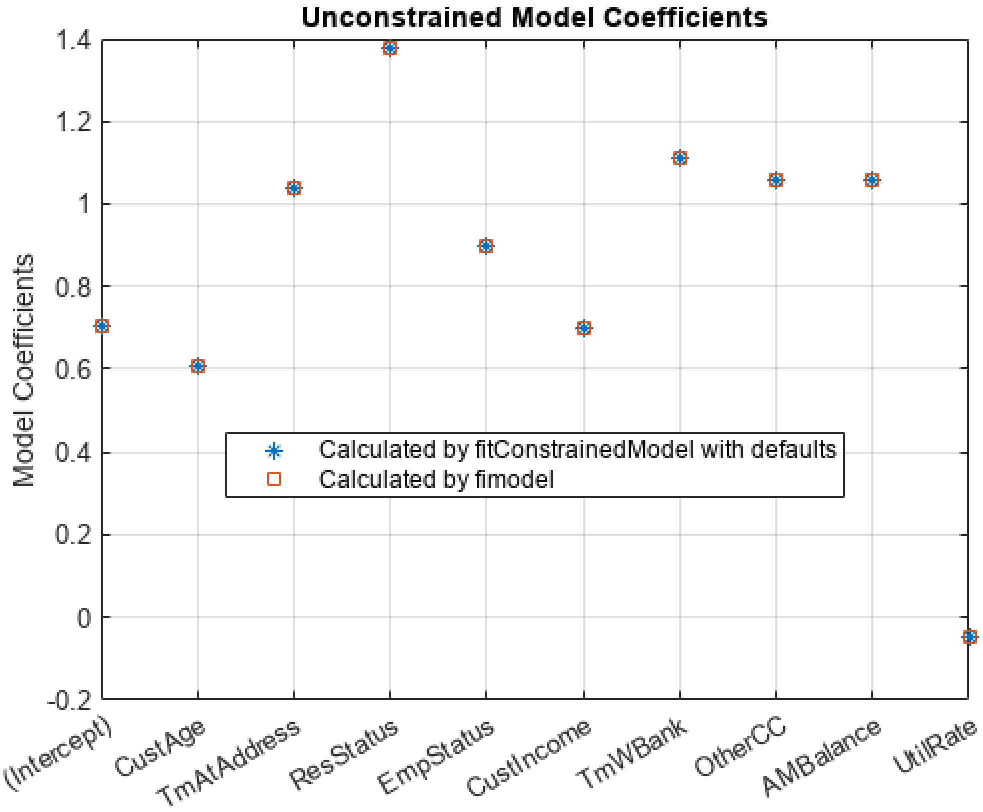
Now, solve the unconstrained problem by using `fitmodel`. Note that `fitmodel` and `fitConstrainedModel` use different solvers. While `fitConstrainedModel` uses `fmincon`, `fitmodel` uses `stepwiseglm` by default. To include all predictors from the start, set the `'VariableSelection'` name-value pair argument of `fitmodel` to `'fullmodel'`.

```
[sc2,mdl2] = fitmodel(sc,'VariableSelection','fullmodel','Display','off');
coeff2 = mdl2.Coefficients.Estimate;
disp(mdl2.Coefficients);
```

```
                   Estimate       SE         tStat        pValue  
                   _________    ________    _________    __________

    (Intercept)      0.70246    0.064039       10.969    5.3719e-28
    CustAge           0.6057     0.24934       2.4292      0.015131
    TmAtAddress       1.0381     0.94042       1.1039       0.26963
    ResStatus         1.3794      0.6526       2.1137      0.034538
    EmpStatus        0.89648     0.29339       3.0556     0.0022458
    CustIncome       0.70179     0.21866       3.2095     0.0013295
    TmWBank           1.1132     0.23346       4.7683    1.8579e-06
    OtherCC           1.0598     0.53005       1.9994      0.045568
    AMBalance         1.0572     0.36601       2.8884     0.0038718
    UtilRate       -0.047597     0.61133    -0.077858       0.93794
```

```
figure
plot(coeff1,'*')
hold on
plot(coeff2,'s')
xticklabels(mdl1.Coefficients.Properties.RowNames)
ylabel('Model Coefficients')
title('Unconstrained Model Coefficients')
```

```
legend({'Calculated by fitConstrainedModel with defaults','Calculated by fimodel'},'Location','be
grid on
```



As both the tables and the plot show, the model coefficients match. You can be confident that this implementation of `fitConstrainedModel` is well calibrated.

Constrained Model

In the constrained model approach, you solve for the values of the coefficients b_i of the logistic model, subject to constraints. The supported constraints are bound, equality, or inequality. The coefficients maximize the likelihood-of-default function defined, for observation i, as:

$$L_i = p(\text{Default}_i)^{y_i} \times (1 - p(\text{Default}_i))^{1 - y_i}$$

where:

- $p(\text{Default}_i) = \dfrac{1}{1 + e^{-b\text{x}_i}}$
- $\boldsymbol{b} = [b_1\, b_2\, .\, .\, .\, b_K]$ is an unknown model coefficient
- $\boldsymbol{x_i} = [x_{\text{i}1}\, x_2\, .\, .\, .\, x_{\text{iK}}]$ is the predictor values at observation i
- y_i is the response value; a value of 1 represents default and a value of 0 represents non-default

This formula is for non-weighted data. When observation i has weight w_i, it means that there are w_i as many observations i. Therefore, the probability that default occurs at observation i is the product of the probabilities of default:

$$p_i \;=\; \underbrace{p(\text{Default}_i)^{y_i} * p(\text{Default}_i)^{y_i} * \ldots * p(\text{Default}_i)^{y_i}}_{w_i\,\text{times}} = p(\text{Default}_i)^{w_i * y_i}$$

Likewise, the probability of non-default for weighted observation i is:

$$\widehat{p}_i \;=\; \underbrace{p(\sim\text{Default}_i)^{1-y_i} * p(\sim\text{Default}_i)^{1-y_i} * \ldots * p(\sim\text{Default}_i)^{1-y_i}}_{w_i\,\text{times}} = (1 - p(\text{Default}_i))^{w_i * (1-y_i)}$$

For weighted data, if there is default at a given observation i whose weight is w_i, it is as if there was a w_i count of that one observation, and all of them either all default, or all non-default. w_i may or may not be an integer.

Therefore, for the weighted data, the likelihood-of-default function for observation i in the first equation becomes

$$L_i = p(\text{Default}_i)^{w_i * y_i} \times (1 - p(\text{Default}_i))^{w_i * (1-y_i)}$$

By assumption, all defaults are independent events, so the objective function is

$$L = L_1 \times L_2 \times \ldots \times L_N$$

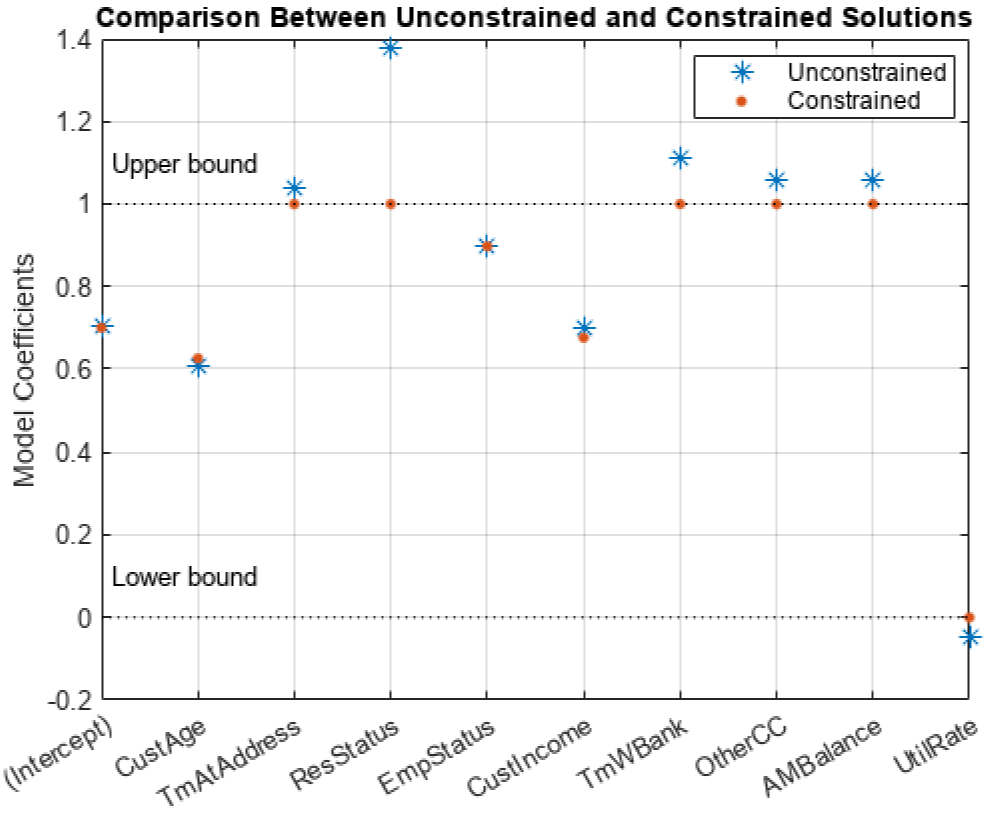
or, in more convenient logarithmic terms:

$$\log(L) = \sum_{i=1}^{N} w_i * [y_i \log(p(\text{Default}_i)) + (1 - y_i)\log(1 - p(\text{Default}_i))]$$

Apply Constraints on the Coefficients

After calibrating the unconstrained model as described in the "Unconstrained Model Using fitConstrainedModel" section, you can solve for the model coefficients subject to constraints. You can choose lower and upper bounds such that $0 \le b_i \le 1, \forall i = 1\ldots K$, except for the intercept. Also, since the customer age and customer income are somewhat correlated, you can also use additional constraints on their coefficients, for example, $|b_{CusAge} - b_{CustIncome}| < 0.1$. The coefficients corresponding to the predictors `'CustAge'` and `'CustIncome'` in this example are b_2 and b_6, respectively.

```
K  = length(sc.PredictorVars);
lb = [-Inf;zeros(K,1)];
ub = [Inf;ones(K,1)];
AIneq = [0 -1 0 0 0 1 0 0 0 0;0 -1 0 0 0 -1 0 0 0 0];
bIneq = [0.05;0.05];
Options = optimoptions('fmincon','SpecifyObjectiveGradient',true,'Display','off');
[sc3,mdl] = fitConstrainedModel(sc,'AInequality',AIneq,'bInequality',bIneq,...
    'LowerBound',lb,'UpperBound',ub,'Options',Options);

figure
plot(coeff1,'*','MarkerSize',8)
hold on
plot(mdl.Coefficients.Estimate,'.','MarkerSize',12)
line(xlim,[0 0],'color','k','linestyle',':')
line(xlim,[1 1],'color','k','linestyle',':')
text(1.1,0.1,'Lower bound')
text(1.1,1.1,'Upper bound')
grid on

xticklabels(mdl.Coefficients.Properties.RowNames)
ylabel('Model Coefficients')
title('Comparison Between Unconstrained and Constrained Solutions')
legend({'Unconstrained','Constrained'},'Location','best')
```

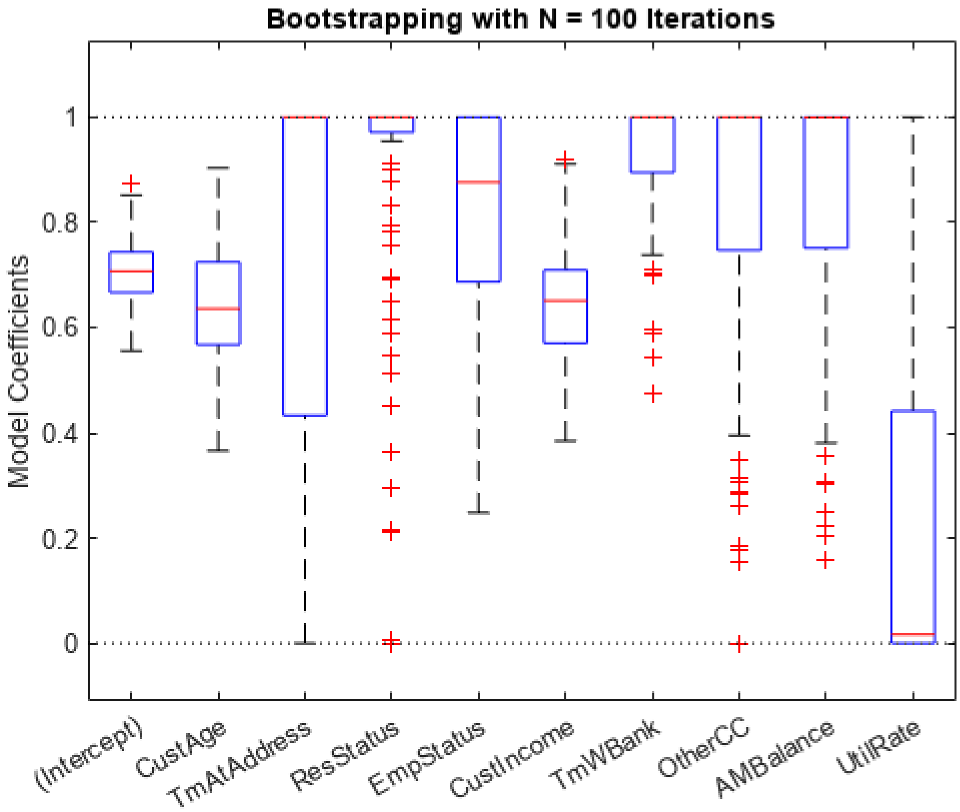


Significance Bootstrapping

For the unconstrained problem, standard formulas are available for computing *p*-values, which you use to evaluate which coefficients are significant and which are to be rejected. However, for the constrained problem, standard formulas are not available, and the derivation of formulas for significance analysis is complicated. A practical alternative is to perform significance analysis through *bootstrapping*.

In the bootstrapping approach, when using `fitConstrainedModel`, you set the name-value argument `'Bootstrap'` to `true` and chose a value for the name-value argument `'BootstrapIter'`. Bootstrapping means that *NIter* samples (with replacement) from the original observations are selected. In each iteration, `fitConstrainedModel` solves for the same constrained problem as the "Constrained Model" section. `fitConstrainedModel` obtains several values (solutions) for each coefficient b_i and you can plot these as a `boxplot` or `histogram`. Using the boxplot or histogram, you can examine the median values to evaluate whether the coefficients are away from zero and how much the coefficients deviate from their means.

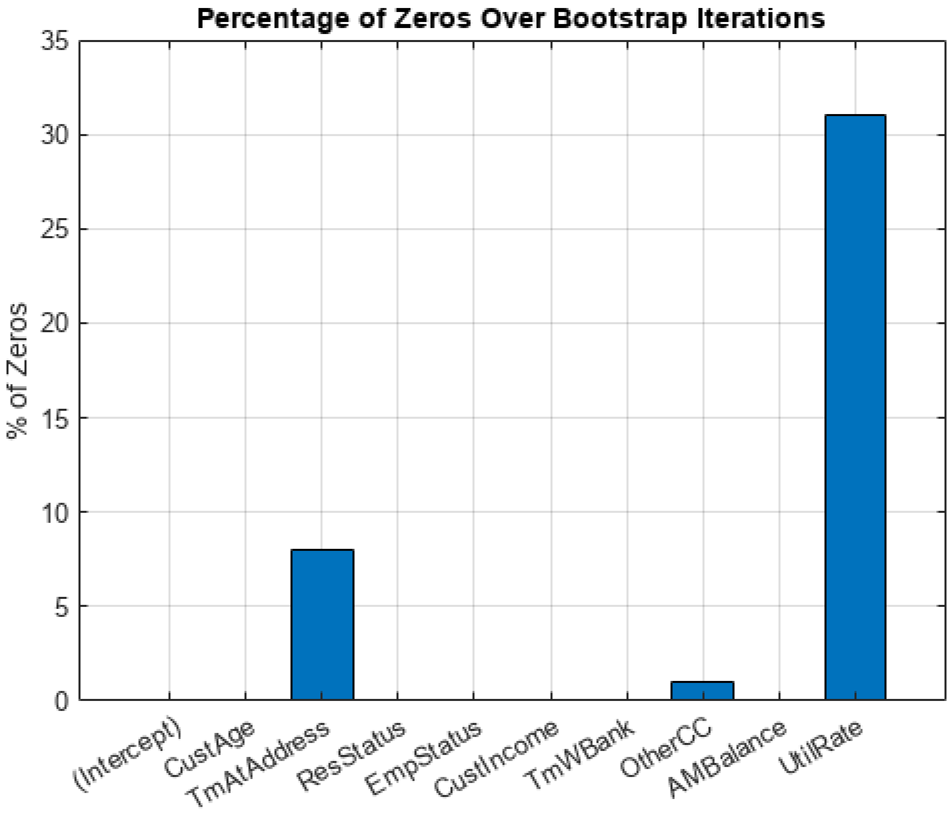
```
lb = [-Inf;zeros(K,1)];
ub = [Inf;ones(K,1)];
AIneq = [0 -1 0 0 0 1 0 0 0 0;0 1 0 0 0 -1 0 0 0 0];
bIneq = [0.05;0.05];
c0 = zeros(K,1);
NIter = 100;
Options = optimoptions('fmincon','SpecifyObjectiveGradient',true,'Display','off');
rng('default')

[sc,mdl] = fitConstrainedModel(sc,'AInequality',AIneq,'bInequality',bIneq,...
    'LowerBound',lb,'UpperBound',ub,'Bootstrap',true,'BootstrapIter',NIter,'Options',Options);

figure
boxplot(mdl.Bootstrap.Matrix,mdl.Coefficients.Properties.RowNames)
hold on
line(xlim,[0 0],'color','k','linestyle',':')
line(xlim,[1 1],'color','k','linestyle',':')
title('Bootstrapping with N = 100 Iterations')
ylabel('Model Coefficients')
```



The solid red lines in the boxplot indicate that the median values and the bottom and top edges are for the 25th and 75th percentiles. The "whiskers" are the minimum and maximum values, not including outliers. The dotted lines represent the lower and upper bound constraints on the coefficients. In this example, the coefficients cannot be negative, by construction.

To help decide which predictors to keep in the model, assess the proportion of times each coefficient is zero.

```
Tol = 1e-6;
figure
bar(100*sum(mdl.Bootstrap.Matrix<= Tol)/NIter)
ylabel('% of Zeros')
title('Percentage of Zeros Over Bootstrap Iterations')
xticklabels(mdl.Coefficients.Properties.RowNames)
grid on
```



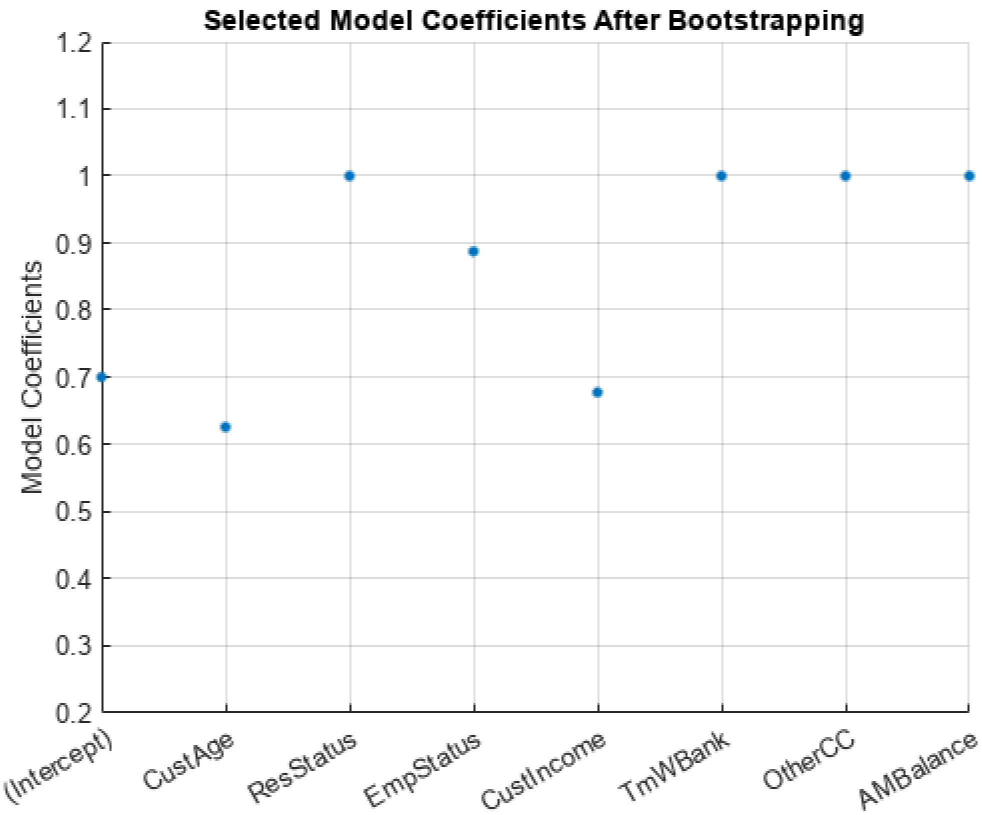
Based on the plot, you can reject `'UtilRate'` since it has the highest number of zero values. You can also decide to reject `'TmAtAddress'` since it shows a peak, albeit small.

Set the Corresponding Coefficients to Zero

To set the corresponding coefficients to zero, set their upper bound to zero and solve the model again using the original data set.

```
ub(3) = 0;
ub(end) = 0;
[sc,mdl] = fitConstrainedModel(sc,'AInequality',AIneq,'bInequality',bIneq,'LowerBound',lb,'Upper
Ind = (abs(mdl.Coefficients.Estimate) <= Tol);
ModelCoeff = mdl.Coefficients.Estimate(~Ind);
ModelPreds = mdl.Coefficients.Properties.RowNames(~Ind)';

figure
hold on
plot(ModelCoeff,'.','MarkerSize',12)
ylim([0.2 1.2])
ylabel('Model Coefficients')
```

```
xticklabels(ModelPreds)
title('Selected Model Coefficients After Bootstrapping')
grid on
```



Set Constrained Coefficients Back Into the creditscorecard

Now that you have solved for the constrained coefficients, use setmodel to set the model's coefficients and predictors. Then you can compute the (unscaled) points.

```
ModelPreds = ModelPreds(2:end);
sc = setmodel(sc,ModelPreds,ModelCoeff);
p = displaypoints(sc);

disp(p)
```

```
      Predictors                Bin                Points  
    _______________    _____________________    _________

    {'CustAge'    }    {'[-Inf,33)'        }     -0.16725
    {'CustAge'    }    {'[33,37)'          }     -0.14811
    {'CustAge'    }    {'[37,40)'          }    -0.065607
    {'CustAge'    }    {'[40,46)'          }    0.044404
    {'CustAge'    }    {'[46,48)'          }      0.21761
    {'CustAge'    }    {'[48,58)'          }      0.23404
    {'CustAge'    }    {'[58,Inf]'         }      0.49029
    {'CustAge'    }    {'<missing>'        }          NaN
    {'ResStatus'  }    {'Tenant'           }    0.0044307
    {'ResStatus'  }    {'Home Owner'       }      0.11932
    {'ResStatus' }    {'Other'            }      0.30048
    {'ResStatus' }    {'<missing>'        }          NaN
    {'EmpStatus' }    {'Unknown'          }    -0.077028
    {'EmpStatus' }    {'Employed'         }      0.31459
    {'EmpStatus' }    {'<missing>'        }          NaN
    {'CustIncome'}    {'[-Inf,29000)'     }     -0.43795
    {'CustIncome'}    {'[29000,33000)'    }    -0.097814
    {'CustIncome'}    {'[33000,35000)'    }     0.053667
    {'CustIncome'}    {'[35000,40000)'    }     0.081921
    {'CustIncome'}    {'[40000,42000)'    }     0.092364
    {'CustIncome'}    {'[42000,47000)'    }      0.23932
    {'CustIncome'}    {'[47000,Inf]'      }      0.42477
    {'CustIncome'}    {'<missing>'        }          NaN
    {'TmWBank'   }    {'[-Inf,12)'        }     -0.15547
    {'TmWBank'   }    {'[12,23)'          }    -0.031077
    {'TmWBank'   }    {'[23,45)'          }    -0.021091
    {'TmWBank'   }    {'[45,71)'          }      0.36703
    {'TmWBank'   }    {'[71,Inf]'         }      0.86888
    {'TmWBank'   }    {'<missing>'        }          NaN
    {'OtherCC'   }    {'No'               }     -0.16832
    {'OtherCC'   }    {'Yes'              }      0.15336
    {'OtherCC'   }    {'<missing>'        }          NaN
    {'AMBalance' }    {'[-Inf,558.88)'    }      0.34418
    {'AMBalance' }    {'[558.88,1254.28)' }    -0.012745
    {'AMBalance' }    {'[1254.28,1597.44)'}    -0.057879
    {'AMBalance' }    {'[1597.44,Inf]'    }     -0.19896
    {'AMBalance' }    {'<missing>'        }          NaN
```

Using the unscaled points, you can follow the remainder of the “Credit Scorecard Modeling Workflow” on page 8-51 to compute scores and probabilities of default and to validate the model.

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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[sc,mdl] = fitConstrainedModel(sc,'LowerBound',2,'UpperBound',100)`

`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 as the comma-separated pair consisting of `'PredictorVars'` and a cell array of character vectors. If you provide predictor variables, then the function updates the `creditscorecard` object property `PredictorsVars`. The order of predictors in the original dataset is enforced, regardless of the order in which

'PredictorVars' is provided. When not provided, the predictors used to create the creditscorecard object (by using creditscorecard) are used.

Data Types: cell

LowerBound — Lower bound
-Inf (default) | scalar | vector

Lower bound, specified as the comma-separated pair consisting of 'LowerBound' and a scalar or a real vector of length N+1, where N is the number of model coefficients in the creditscorecard object.

Data Types: double

UpperBound — Upper bound
Inf (default) | scalar | vector

Upper bound, specified as the comma-separated pair consisting of 'UpperBound' and a scalar or a real vector of length N+1, where N is the number of model coefficients in the creditscorecard object.

Data Types: double

AInequality — Matrix of linear inequality constraints
[] (default) | matrix

Matrix of linear inequality constraints, specified as the comma-separated pair consisting of 'AInequality' and a real M-by-N+1 matrix, where M is the number of constraints and N is the number of model coefficients in the creditscorecard object.

Data Types: double

bInequality — Vector of linear inequality constraints
[] (default) | vector

Vector of linear inequality constraints, specified as the comma-separated pair consisting of 'bInequality' and a real M-by-1 vector, where M is the number of constraints.

Data Types: double

AEquality — Matrix of linear equality constraints
[] (default) | matrix

Matrix of linear equality constraints, specified as the comma-separated pair consisting of 'AEquality' and a real M-by-N+1 matrix, where M is the number of constraints and N is the number of model coefficients in the creditscorecard object.

Data Types: double

bEquality — Vector of linear equality constraints
[] (default) | vector

Vector of linear equality constraints, specified as the comma-separated pair consisting of 'bEquality' and a real M-by-1 vector, where M is the number of constraints.

Data Types: double

Bootstrap — Indicator that bootstrapping defines the solution accuracy

false (default) | logical with a value of true or false

Indicator that bootstrapping defines the solution accuracy, specified as the comma-separated pair consisting of 'Bootstrap' and a logical with a value of true or false.

Data Types: logical

BootstrapIter — Number of bootstrapping iterations

100 (default) | positive integer

Number of bootstrapping iterations, specified as the comma-separated pair consisting of 'BootstrapIter' and a positive integer.

Data Types: double

Options — optimoptions object

optimoptions('fmincon','SpecifiedObjectiveGradient',true,'Display','off') (default) | object

optimoptions object, specified as the comma-separated pair consisting of 'Options' and an optimoptions object. You can create the object by using optimoptions from Optimization Toolbox.

Data Types: object

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 that 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 a GeneralizedLinearModel object containing the fitted model. For more information on a GeneralizedLinearModel object, see GeneralizedLinearModel.

Note If you specify the optional WeightsVar argument when creating a creditscorecard object, then mdl uses the weighted counts with stepwiseglm and fitglm.

The mdl structure has the following fields:

- Coefficients is a table in which the RowNames property contains the names of the model coefficients and has a single column, 'Estimate', containing the solution.
- Bootstrap exists when 'Bootstrap' is set to true, and has two fields:
  - CI contains the 95% confidence interval for the solution.
  - Matrix an NIter-by-N matrix of coefficients, where NIter is the number of bootstrap iterations and N is the number of model coefficients.

- Solver has three fields:
  - Options additional information on the algorithm and solution.
  - ExitFlag contains an integer that codes the reason why the solver stopped. For more information, see fmincon.
  - Output is a structure with additional information on the optimization process.

More About

Model Coefficients

When you use fitConstrainedModel to solve for the model coefficients, the function solves for the same number of parameters as predictor variables you specify, plus one additional coefficient for the intercept.

The first coefficient corresponds to the intercept. If you provide predictor variables using the 'PredictorVars' optional input argument, then fitConstrainedModel updates the creditscorecard object property PredictorsVars. The order of predictors in the original dataset is enforced, regardless of the order in which 'PredictorVars' is provided. When not provided, the predictors used to create the creditscorecard object (by using creditscorecard) are used.

Calibration

The constrained model is first calibrated such that, when unconstrained, the solution is identical, within a certain tolerance, to the solution given by fitmodel, with the'fullmodel' choice for the name-value argument 'VariableSelection'.

As an exercise, you can test the calibration by leaving all name-value parameters of fitConstrainedModel to their default values. The solutions are identical to within a 10^{-6} to 10^{-5} tolerance.

Calibration with Weights and Missing Data

If the credit scorecard data contains observation weights, the fitConstrainedModel function uses the weights to calibrate the model coefficients.

For credit scorecard data with no missing data and no weights, the likelihood function for observation i is

$$L_i = p(^{\text{Default}_i} \times (^1$$

$$\text{where } p(\text{Default}_i) = \frac{1}{(1 + e^{-bx_i})}$$

where:

- $b = [b_1\ b_2 \ldots b_K]$ is for unknown model coefficients
- $x_i = [x_{i1}\ x_{i2} \ldots x_{iK}]$ is the predictor values at observation i
- y_i is the response value of 1 (the default) or a value of 0.

When observation i has weight w_i, it means that there are w_i observations. Because of the independence of defaults between observations, the probability that there is default at observation i is the product of the probabilities of default

$$p_i = p(^{\mathrm{Default}_i} * p(^{\mathrm{Default}_i} * \ldots * p(^{\mathrm{Default}_i} = p(^{\mathrm{Default}_i}$$

$$w_i \text{ times}$$

Likewise, the probability of non-default for weighted observation *i* is

$$\widehat{p}_i = p(^{\sim} * p(^{\sim} * \ldots * p(^{\sim} = (1 - p(^{\mathrm{Default}_i}$$

$$w_i \text{ times}$$

For weighted data, if there is default at a given observation *i* whose weight is w_i, it is as if there was w_i defaults of that one observation, and all of them either all default, or all non-default. w_i may or may not be an integer. Therefore, the likelihood function for observation *i* becomes

$$L_i = p(\mathrm{Default}_i)^{w_i * y_i} \times (1 - p(\mathrm{Default}_i))^{w_i * (1 - y_i)}$$

Likewise, for data with missing observations (`NaN`, `<undefined>`, or “`Missing`”), the model is calibrated by comparing the unconstrained case with results given by `fitglm`. Where the data contains missing observations, the WOE input matrix has `NaN` values. The `NaN` values do not pose any issue for `fitglm` (unconstrained), or `fmincon` (constrained). The only edge case is if all observations of a given predictor are missing, in which case, that predictor is discarded from the model.

Bootstrapping

Bootstrapping is a method for estimating the accuracy of the solution obtained after iterating the objective function `NI`ter times.

When `'Bootstrap'` is set to `true`, the `fitConstrainedModel` function performs sampling with replacement of the WOE values and is passed to the objective function. At the end of the iterative process, the solutions are stored in a `NI`ter-by-`N+1` matrix, where `N` is the number of model coefficients.

The 95% confidence interval (CI) returned in the output structure `mdl.Bootstrap` contains the values of the coefficients at the 25th and 97.5th percentiles.

Models

A logistic regression model is used in the `creditscorecard` object.

For the model, the probability of being “Bad” is given by `ProbBad = exp(-s) / (1 + exp(-s))`.

Version History

Introduced in R2019a

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

`fitmodel` | `creditscorecard` | `autobinning` | `bininfo` | `predictorinfo` | `modifypredictor` | `plotbins` | `modifybins` | `bindata` | `displaypoints` | `formatpoints` | `score` | `stepwiseglm` | `fitglm` | `fmincon` | `setmodel` | `probdefault` | `validatemodel` | `GeneralizedLinearModel`

Topics

“Case Study for Credit Scorecard Analysis” on page 8-70
“Credit Scorecard Modeling with Missing Values” on page 8-56
“Troubleshooting Credit Scorecard Results” on page 8-63
“Credit Scorecard Modeling Workflow” on page 8-51
“About Credit Scorecards” on page 8-47
“Credit Scorecard Modeling Using Observation Weights” on page 8-54
“What Are Generalized Linear Models?”

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 15-1744.

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: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustI
        NumericPredictors: {'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBalance'  'Uti
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
```

```
          BinMissingData: 0
                   IDVar: 'CustID'
           PredictorVars: {'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIncome'  'Tr
                    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-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 + AMBala
    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:
    logit(status) ~ 1 + CustAge + ResStatus + EmpStatus + CustIncome + TmWBank + OtherCC + AMBala
    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
    {'CustAge'   }
    {'ResStatus' }
    {'EmpStatus' }
    {'CustIncome'}
    {'TmWBank'   }
    {'OtherCC'   }
    {'AMBalance' }
    {'UtilRate'  }
```

```
ModelCoefficients = mdl.Coefficients.Estimate
```

```
ModelCoefficients = 9×1

    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=41×3 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
    {'CustAge'   }    {'<missing>'   }          NaN
    {'ResStatus' }    {'Tenant'      }    -0.043826
    {'ResStatus' }    {'Home Owner'  }      0.11442
    {'ResStatus' }    {'Other'       }      0.36394
    {'ResStatus' }    {'<missing>'   }          NaN
    {'EmpStatus' }    {'Unknown'     }    -0.088843
    {'EmpStatus' }    {'Employed'    }      0.30193
    {'EmpStatus' }    {'<missing>'   }          NaN
    {'CustIncome'}    {'[-Inf,29000)'}     -0.46956
      ⋮
```

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: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustI
        NumericPredictors: {'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBalance'  'Uti
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
           BinMissingData: 0
                    IDVar: 'CustID'
            PredictorVars: {'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIncome'  'Tr
                     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=10×4 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=10×4 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-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

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
    {'CustAge'   }
    {'EmpStatus' }
    {'CustIncome'}
    {'TmWBank'   }
    {'AMBalance' }
```

```
ModelCoefficients = mdl.Coefficients.Estimate
```

```
ModelCoefficients = 6×1

    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=30×3 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
    {'CustAge'   }    {'<missing>'    }          NaN
    {'EmpStatus' }    {'Unknown'      }    -0.035716
    {'EmpStatus' }    {'Employed'     }      0.35417
    {'EmpStatus' }    {'<missing>'    }          NaN
    {'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
      ⋮
```

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`.

More About

Workflows for Using `setmodel`

When using `setmodel`, there are two possible workflows to set the final model predictors and model coefficients into a `creditscorecard` object.

The first workflow is:

- Use `fitmodel` to get the optional output argument `mdl`. This is a `GeneralizedLinearModel` object and you can add and remove terms, or modify the parameters of the stepwise procedure. Only linear terms can be in the model (no interactions or any other higher-order terms).
- Once the `GeneralizedLinearModel` object is satisfactory, set the final model predictors and model coefficients into the `creditscorecard` object using the `setmodel` input arguments for `ModelPredictors` and `ModelCoefficients`.

An alternate workflow is:

- Obtain the Weight of Evidence (WOE) data using `bindata`. Use the `'WOEModelInput'` option for the `'OutputType'` name-value pair argument in `bindata` to ensure that:
  - The predictors data is transformed to WOE.
  - Only predictors whose bins have finite WOE values are included.
  - The response variable is placed in the last column.
  - The response variable is mapped (“Good” is `1` and “Bad” is `0`).
- Use the data from the previous step to fit a linear logistic regression model (only linear terms in the model, no interactions, or any other higher-order terms). See, for example, `stepwiseglm`.
- Once the `GeneralizedLinearModel` object is satisfactory, set the final model predictors and model coefficients into the `creditscorecard` object using the `setmodel` input arguments for `ModelPredictors` and `ModelCoefficients`.

Version History

Introduced in R2014b

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

`creditscorecard` | `autobinning` | `bininfo` | `predictorinfo` | `modifypredictor` | `plotbins` | `modifybins` | `bindata` | `displaypoints` | `formatpoints` | `score` | `stepwiseglm` | `fitglm` | `fitmodel` | `probdefault` | `validatemodel` | `GeneralizedLinearModel`

Topics

"Case Study for Credit Scorecard Analysis" on page 8-70
"Troubleshooting Credit Scorecard Results" on page 8-63
"Credit Scorecard Modeling Workflow" on page 8-51
"About Credit Scorecards" on page 8-47

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: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIn
        NumericPredictors: {'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBalance'  'Uti
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
           BinMissingData: 0
                    IDVar: 'CustID'
            PredictorVars: {'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIncome'  'Tr
                     Data: [1200x11 table]
```

Perform automatic binning.

```
sc = autobinning(sc);
```

Show the bin information for `'CustAge'`.

```
bininfo(sc,'CustAge')
```

```
ans=8×6 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=10×1 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=10×1 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=10×1 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=10×1 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: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustI
        NumericPredictors: {'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBalance'  'Uti
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
           BinMissingData: 0
                    IDVar: 'CustID'
            PredictorVars: {'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIncome'  'T
                     Data: [1200x11 table]
```

Perform automatic binning.

```
sc = autobinning(sc);
```

Show the bin information for `'CustAge'`.

```
bininfo(sc,'CustAge')
```

```
ans=8×6 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=10×1 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=10×11 table
    CustID    CustAge     TmAtAddress    ResStatus    EmpStatus    CustIncome    TmWBank    Othe
    ______    ________    ___________    _________    _________    __________    ________    ____

     101      -0.39568     -0.087767     -0.095564      0.2418     -0.011271      0.76889    0.05
     102       0.62245       0.14288      0.019329    -0.19947       0.20579     -0.13107    -0.2
     103       0.62245       0.02263      0.019329      0.2418       0.47972     -0.12109    0.05
     104       0.62245       0.02263     -0.095564      0.2418       0.47972     -0.12109    0.05
     105      -0.42622       0.02263      0.019329      0.2418      -0.06843      0.76889    0.05
     106       0.62245       0.02263      0.019329    -0.19947       0.20579     -0.13107    0.05
     107       0.21378     -0.087767     -0.095564       0.2418       0.47972       0.26704     0.05
     108      -0.26411     -0.087767      0.019329     -0.19947      -0.29217      -0.13107     0.05
     109       0.21378     -0.087767     -0.095564       0.2418      -0.026696     -0.13107     0.05
     110       0.21378     -0.087767      0.019329       0.2418       0.20579      -0.13107     0.05
```

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=10×1 table
    CustAge 
    ________

    -0.39568
     0.62245
     0.62245
     0.62245
    -0.42622
     0.62245
     0.21378
    -0.26411
     0.21378
     0.21378
```

Bin Additional "Test" Data When Using the `'BinMissingData'` Option

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` with missing values. The variables `CustAge` and `ResStatus` have missing values.

```
load CreditCardData.mat
head(dataMissing,5)
```

```
    CustID    CustAge    TmAtAddress     ResStatus     EmpStatus    CustIncome    TmWBank    Othe
    ______    _______    ___________    ___________    _________    __________    _______    ____

      1          53          62         <undefined>    Unknown        50000         55         Ye
      2          61          22         Home Owner     Employed       52000         25         Ye
      3          47          30         Tenant         Employed       37000         61         No
      4         NaN          75         Home Owner     Employed       53000         20         Ye
      5          68          56         Home Owner     Employed       53000         14         Ye
```

Use `creditscorecard` with the name-value argument `'BinMissingData'` set to `true` to bin the missing numeric or categorical data in a separate bin. Apply automatic binning.

```
sc = creditscorecard(dataMissing,'IDVar','CustID','BinMissingData',true);
sc = autobinning(sc);

disp(sc)
```

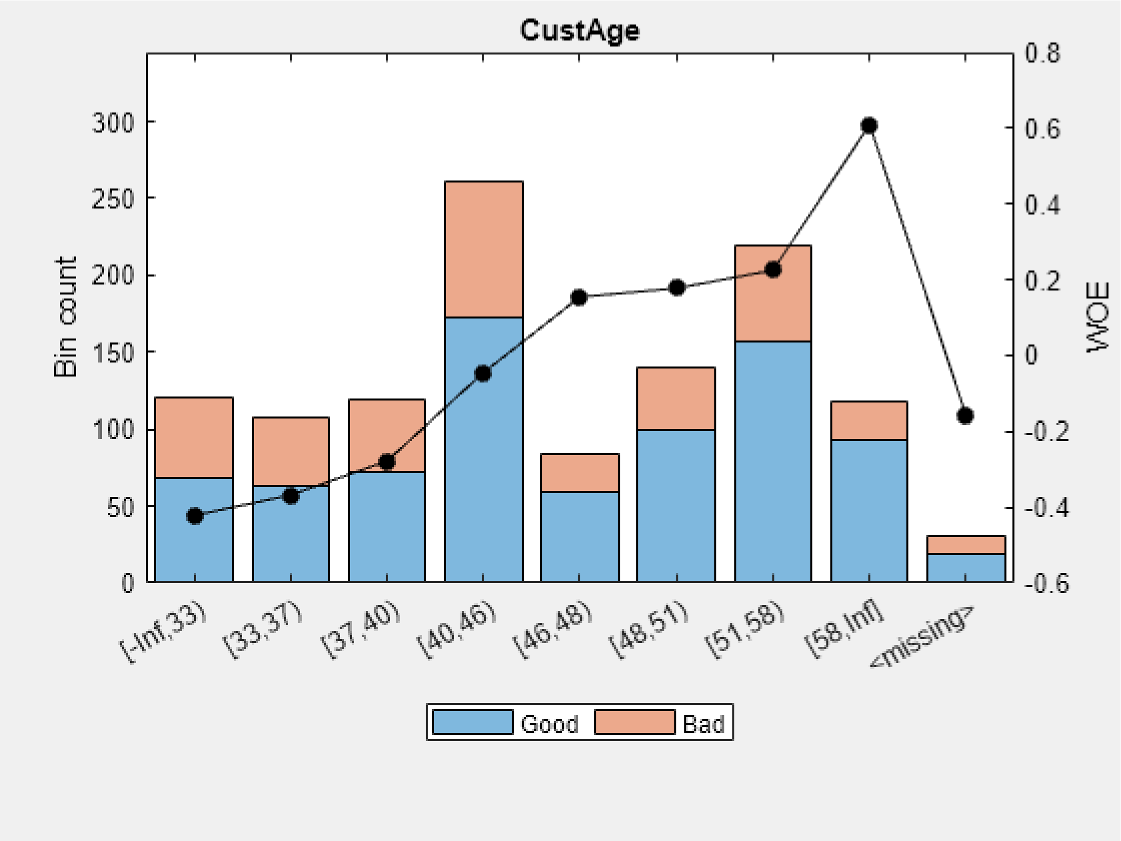
```
  creditscorecard with properties:

                GoodLabel: 0
              ResponseVar: 'status'
              WeightsVar: ''
                VarNames: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIn
       NumericPredictors: {'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBalance'  'Uti
   CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
          BinMissingData: 1
                   IDVar: 'CustID'
           PredictorVars: {'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIncome'  'Tr
                    Data: [1200x11 table]
```

Display and plot bin information for numeric data for `'CustAge'` that includes missing data in a separate bin labelled `<missing>`.

```
[bi,cp] = bininfo(sc,'CustAge');
disp(bi)
```

```
         Bin         Good    Bad     Odds       WOE       InfoValue 
    _____________    ____    ___    ______    ________    __________

    {'[-Inf,33)'}     69     52     1.3269    -0.42156      0.018993
    {'[33,37)'  }     63     45        1.4    -0.36795      0.012839
    {'[37,40)'  }     72     47     1.5319     -0.2779     0.0079824
    {'[40,46)'  }    172     89     1.9326    -0.04556     0.0004549
    {'[46,48)'  }     59     25       2.36     0.15424     0.0016199
    {'[48,51)'  }     99     41     2.4146     0.17713     0.0035449
    {'[51,58)'  }    157     62     2.5323     0.22469     0.0088407
    {'[58,Inf]' }     93     25       3.72     0.60931      0.032198
    {'<missing>'}     19     11     1.7273    -0.15787    0.00063885
    {'Totals'   }    803    397     2.0227         NaN      0.087112
```

```
plotbins(sc,'CustAge')
```

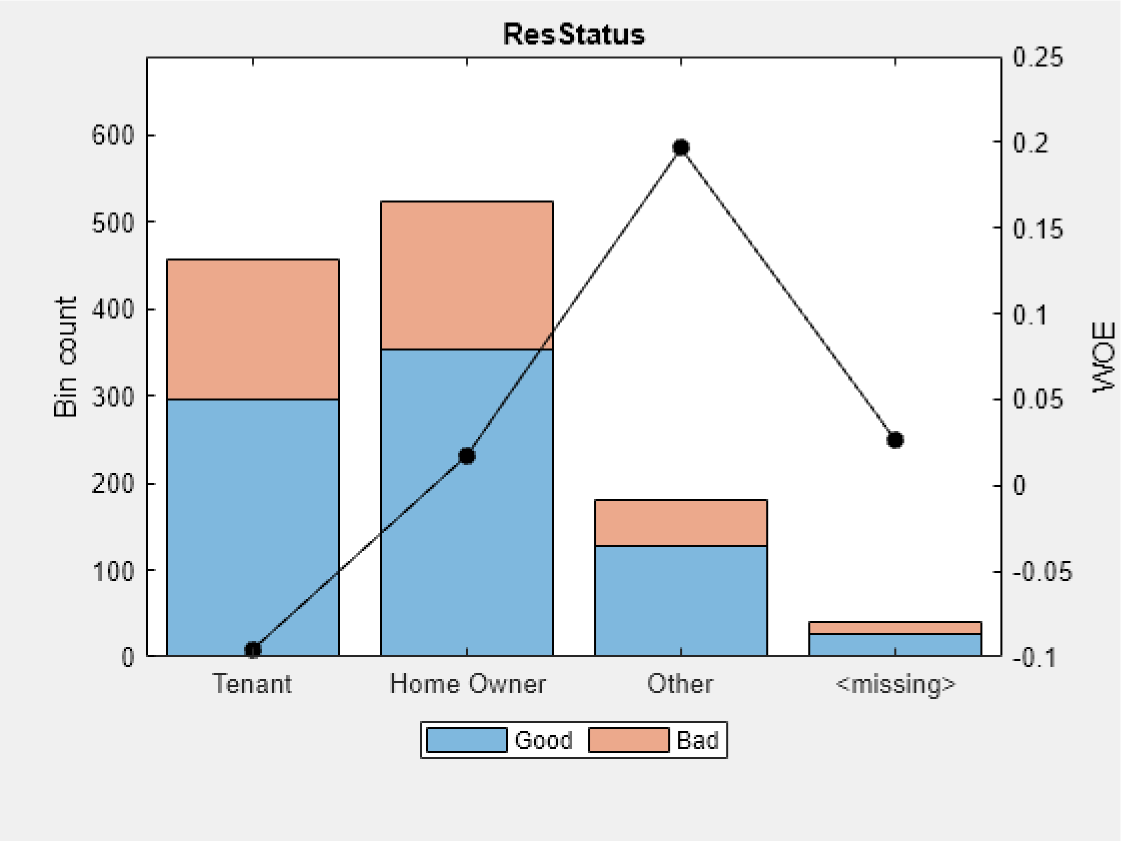


Display and plot bin information for categorical data for `'ResStatus'` that includes missing data in a separate bin labelled `<missing>`.

```
[bi,cg] = bininfo(sc,'ResStatus');
disp(bi)
```

```
         Bin           Good    Bad     Odds        WOE       InfoValue 
    ______________     ____    ___    ______    _________    __________

    {'Tenant'     }    296     161    1.8385    -0.095463     0.0035249
    {'Home Owner'}     352     171    2.0585     0.017549    0.00013382
    {'Other'      }    128      52    2.4615      0.19637     0.0055808
    {'<missing>'  }     27      13    2.0769     0.026469    2.3248e-05
    {'Totals'     }    803     397    2.0227          NaN     0.0092627
```

```
plotbins(sc,'ResStatus')
```



For the 'CustAge' and 'ResStatus' predictors, there is missing data (NaNs and <undefined>) in the training data, and the binning process estimates a WOE value of -0.15787 and 0.026469 respectively for missing data in these predictors, as shown above.

For the purpose of illustration, take a few rows from the original data as test data and introduce some missing data.

```
tdata = dataMissing(11:14,:);
tdata.CustAge(1) = NaN;
tdata.TmAtAddress(2) = NaN;
tdata.ResStatus(3) = '<undefined>';
tdata.EmpStatus(4) = '<undefined>';
disp(tdata)
```

```
    CustID    CustAge    TmAtAddress     ResStatus      EmpStatus      CustIncome    TmWBank    Ot
    ______    _______    ___________    ___________    ___________    __________    _______    __

      11        NaN          24         Tenant         Unknown          34000          44
      12         48         NaN         Other          Unknown          44000          14
      13         65          63         <undefined>    Unknown          48000           6
      14         44          75         Other          <undefined>      41000          35
```

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');
disp(bdata)
```

```
    CustID    CustAge     TmAtAddress    ResStatus    EmpStatus    CustIncome    TmWBank     Othe
    ______    ________    ___________    _________    _________    __________    ________    ___

      11    -0.15787      0.02263    -0.095463    -0.19947     -0.06843    -0.12109    0.05
      12     0.17713          NaN      0.19637    -0.19947      0.20579    -0.13107    0.05
      13     0.60931      0.02263     0.026469    -0.19947      0.47972    -0.25547    -0.2
      14    -0.04556      0.02263      0.19637         NaN    -0.011271    -0.12109    -0.2
```

For the `'CustAge'` and `'ResStatus'` predictors, because there is missing data in the training data, the missing values in the test data get mapped to the WOE value estimated for the `<missing>` bin. Therefore, a missing value for `'CustAge'` is replaced with `-0.15787`, and a missing value for `'ResStatus'` is replaced with `0.026469`.

For `'TmAtAddress'` and `'EmpStatus'`, the training data has no missing values, therefore there is no bin for missing data, and there is no way to estimate a WOE value for missing data. Therefore, for these predictors, the WOE transformation leaves missing values as missing (that is, sets a WOE value of `NaN`).

These rules apply when `'OutputType'` is set to `'WOE'` or `'WOEModelInput'`. The rationale is that if a data-based WOE value exists for missing data, it should be used for the WOE transformation and for subsequent steps (for example, fitting a logistic model or scoring).

On the other hand, when `'OutputType'` is set to `'BinNumber'` or `'Categorical'`, `bindata` leaves missing values as missing, since this allows you to subsequently treat the missing data as you see fit.

For example, when `'OutputType'` is set to `'BinNumber'`, missing values are set to `NaN`:

```
bdata = bindata(sc,tdata,'OutputType','BinNumber');
disp(bdata)
```

```
    CustID    CustAge    TmAtAddress    ResStatus    EmpStatus    CustIncome    TmWBank    Other
    ______    _______    ___________    _________    _________    __________    _______    ____

      11        NaN            2             1            1           3            3          2
      12          6          NaN             3            1           6            2          2
      13          8            2           NaN            1           7            1          1
      14          4            2             3          NaN           5            3          1
```

And when `'OutputType'` is set to `'Categorical'`, missing values are set to `'<undefined>'`:

```
bdata = bindata(sc,tdata,'OutputType','Categorical');
disp(bdata)
```

```
    CustID      CustAge      TmAtAddress      ResStatus      EmpStatus      CustIncome       TmWBa
    ______    ___________    ___________    ___________    ___________    _____________    _____

      11      <undefined>    [23,83)        Tenant         Unknown        [33000,35000)    [23,45
      12      [48,51)        <undefined>    Other          Unknown        [42000,47000)    [12,23
      13      [58,Inf]       [23,83)        <undefined>    Unknown        [47000,Inf]      [-Inf
      14      [40,46)        [23,83)        Other          <undefined>    [40000,42000)    [23,45
```

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: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIn
        NumericPredictors: {'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBalance'  'Uti
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
           BinMissingData: 0
                    IDVar: 'CustID'
            PredictorVars: {'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIncome'  'Tr
                     Data: [1200x11 table]
```

Perform automatic binning.

```
sc = autobinning(sc);
```

Show the bin information for `'CustAge'`.

```
bininfo(sc,'CustAge')
```

```
ans=8×6 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=10×1 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=10×1 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=10×1 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             108987  table              
  data       1200x11              84603  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=10×1 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             99167  table              
  data       1200x11             84603  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
```

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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `bdata = bindata(sc,'OutputType','WOE','ResponseFormat','Mapped')`

OutputType — Output format

`'BinNumber'` (default) | character vector with values `'BinNumber'`, `'Categorical'`, `'WOE'`

Output format, specified as the comma-separated pair consisting of `'OutputType'` and 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 as the comma-separated pair consisting of `'ResponseFormat'` and 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'`.

Version History

Introduced in R2014b

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

`creditscorecard` | `autobinning` | `bininfo` | `predictorinfo` | `modifypredictor` | `plotbins` | `modifybins` | `fitmodel` | `displaypoints` | `formatpoints` | `score` | `setmodel` | `probdefault` | `validatemodel`

Topics

“Case Study for Credit Scorecard Analysis” on page 8-70
“Credit Scorecard Modeling with Missing Values” on page 8-56
“Troubleshooting Credit Scorecard Results” on page 8-63
“Credit Scorecard Modeling Workflow” on page 8-51
“About Credit Scorecards” on page 8-47

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: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustI
        NumericPredictors: {'CustID'  'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBala
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
           BinMissingData: 0
                    IDVar: ''
```

```
         PredictorVars: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustI
                  Data: [1200x11 table]
```

Use `bininfo` to display the autobinned data.

```
[bi, cp] = bininfo(sc, 'CustIncome')
```

```
bi=8×6 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 = 6×1

       29000
       33000
       35000
       40000
       42000
       47000
```

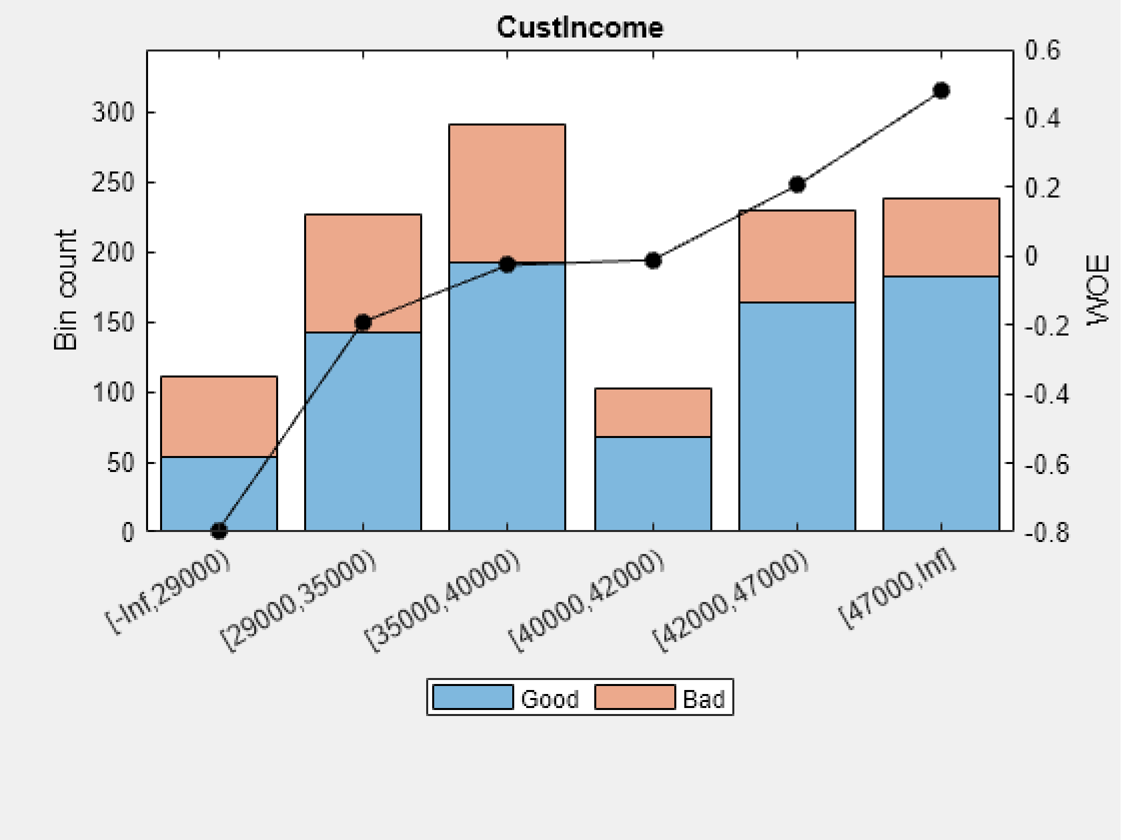
Manually remove the second cut point (the boundary between the second and third bins) to merge bins two and three. Use the `modifybins` function to update the scorecard and then display updated bin information.

```
cp(2) = [];
sc = modifybins(sc,'CustIncome','CutPoints',cp);
bi = bininfo(sc,'CustIncome')
```

```
bi=7×6 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: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustI
        NumericPredictors: {'CustID'  'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBala
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
           BinMissingData: 0
                    IDVar: ''
            PredictorVars: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustI
```

```
                        Data: [1200x11 table]
```

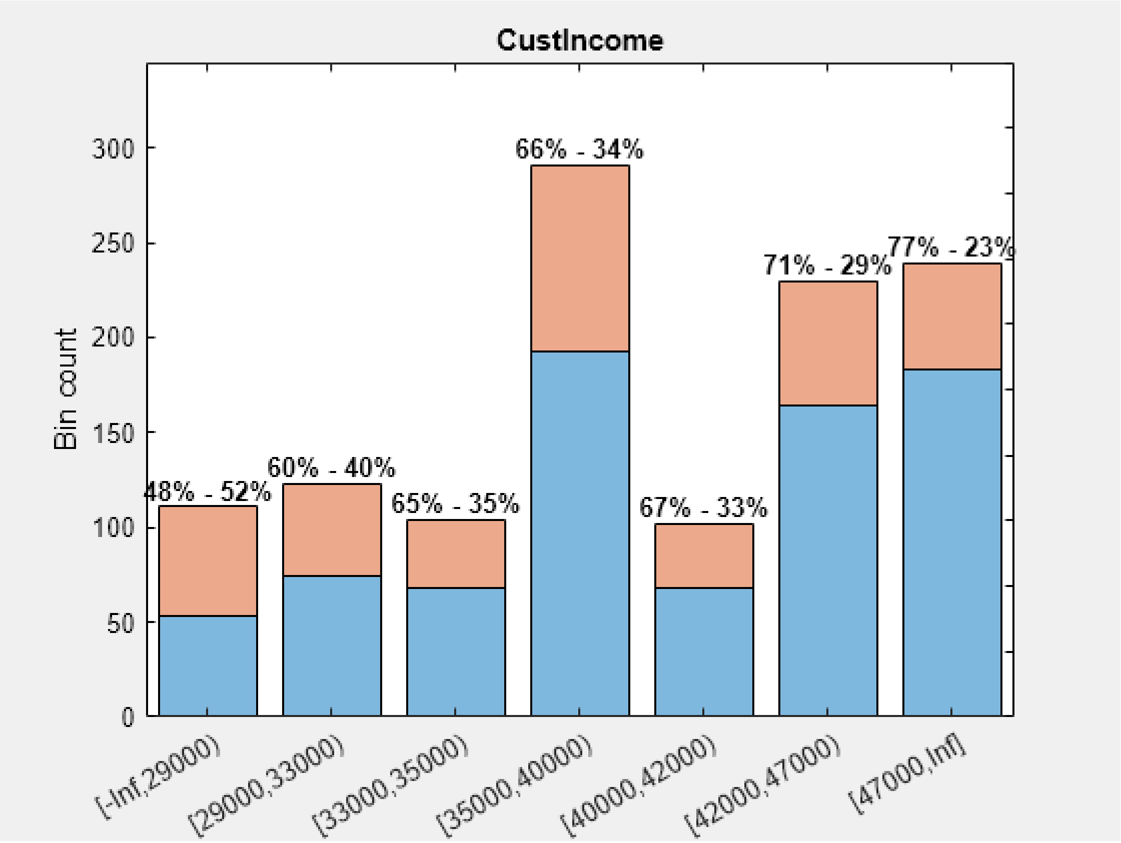
Use `bininfo` to display the autobinned data.

```
[bi, cp] = bininfo(sc, 'CustIncome')
```

```
bi=8×6 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 = 6×1

       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')
```



Plot Bin Information When Using Missing Data

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` with missing values.

```
load CreditCardData.mat
head(dataMissing,5)
```

```
    CustID    CustAge    TmAtAddress     ResStatus     EmpStatus    CustIncome    TmWBank    Othe
    ______    _______    ___________    ___________    _________    __________    _______    ____

      1          53          62         <undefined>    Unknown        50000         55         Ye
      2          61          22         Home Owner     Employed       52000         25         Ye
      3          47          30         Tenant         Employed       37000         61         No
      4         NaN          75         Home Owner     Employed       53000         20         Ye
      5          68          56         Home Owner     Employed       53000         14         Ye
```

```
fprintf('Number of rows: %d\n',height(dataMissing))
```

```
Number of rows: 1200
```

```
fprintf('Number of missing values CustAge: %d\n',sum(ismissing(dataMissing.CustAge)))
```

```
Number of missing values CustAge: 30
```

```
fprintf('Number of missing values ResStatus: %d\n',sum(ismissing(dataMissing.ResStatus)))
```

```
Number of missing values ResStatus: 40
```

Use `creditscorecard` with the name-value argument `'BinMissingData'` set to `true` to bin the missing numeric or categorical data in a separate bin.

```
sc = creditscorecard(dataMissing,'IDVar','CustID','BinMissingData',true);
sc = autobinning(sc);

disp(sc)
```

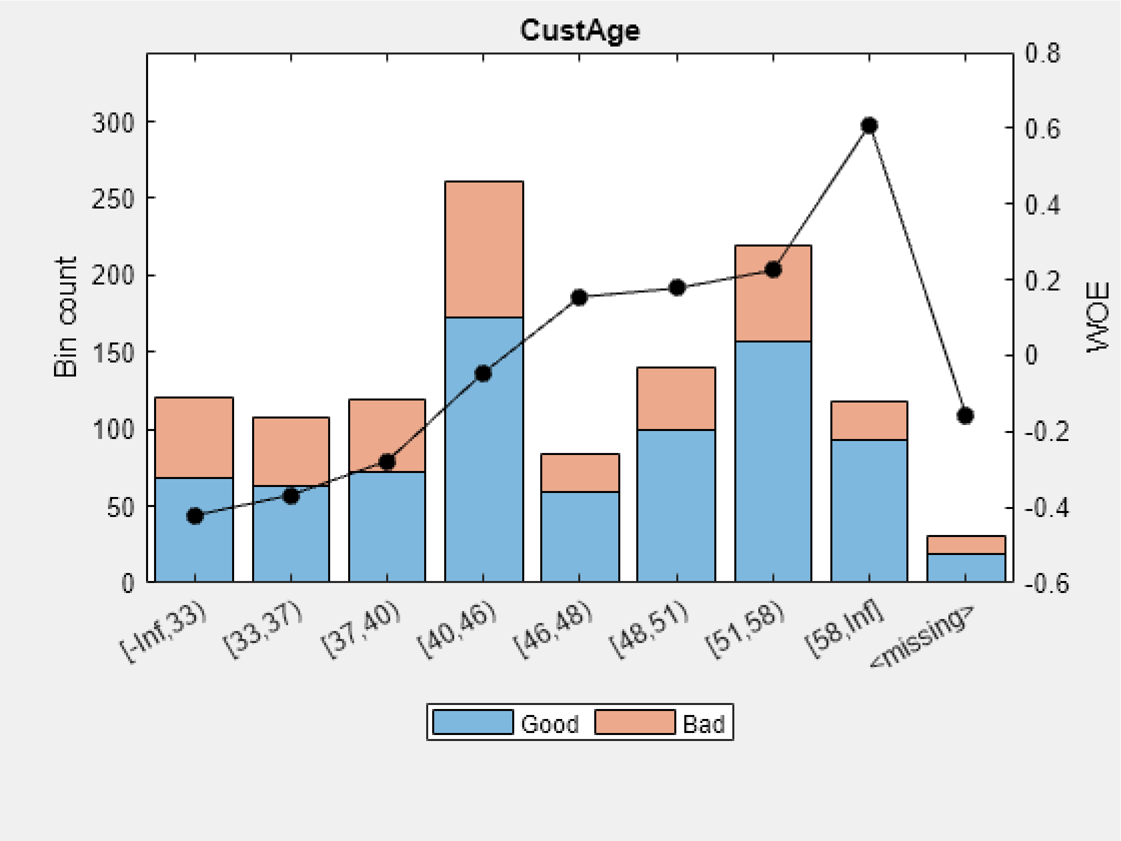
```
  creditscorecard with properties:

                GoodLabel: 0
              ResponseVar: 'status'
               WeightsVar: ''
                 VarNames: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIn
        NumericPredictors: {'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBalance'  'Uti
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
           BinMissingData: 1
                    IDVar: 'CustID'
            PredictorVars: {'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIncome'  'Tr
                     Data: [1200x11 table]
```

Display and plot bin information for numeric data for `'CustAge'` that includes missing data in a separate bin labelled `<missing>`.

```
[bi,cp] = bininfo(sc,'CustAge');
disp(bi)
```

```
         Bin          Good    Bad     Odds        WOE       InfoValue 
    ______________    ____    ___    ______    ________    __________

    {'[-Inf,33)'}      69     52     1.3269    -0.42156      0.018993
    {'[33,37)'  }      63     45        1.4    -0.36795      0.012839
    {'[37,40)'  }      72     47     1.5319     -0.2779     0.0079824
    {'[40,46)'  }     172     89     1.9326    -0.04556     0.0004549
    {'[46,48)'  }      59     25       2.36     0.15424     0.0016199
    {'[48,51)'  }      99     41     2.4146     0.17713     0.0035449
    {'[51,58)'  }     157     62     2.5323     0.22469     0.0088407
    {'[58,Inf]' }      93     25       3.72     0.60931      0.032198
    {'<missing>'}      19     11     1.7273    -0.15787    0.00063885
    {'Totals'   }     803    397     2.0227         NaN      0.087112
```

```
plotbins(sc,'CustAge')
```

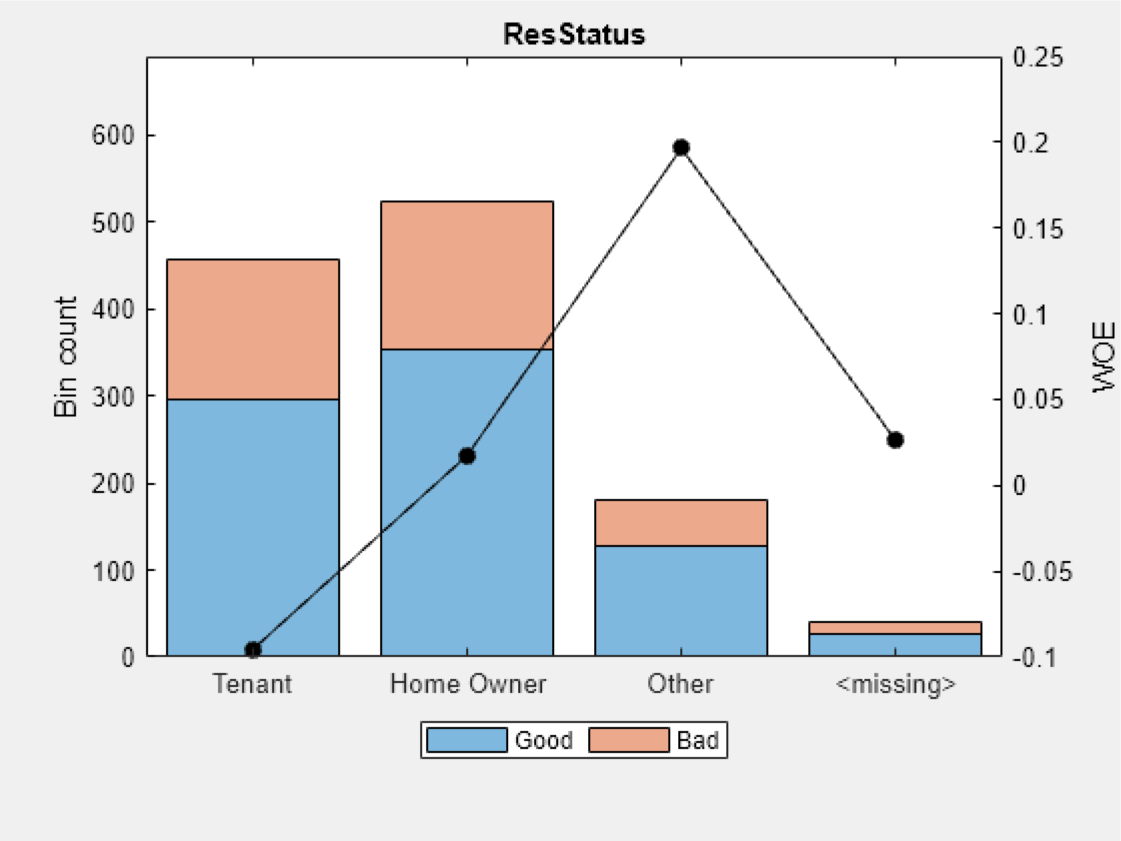


Display and plot bin information for categorical data for `'ResStatus'` that includes missing data in a separate bin labelled `<missing>`.

```
[bi,cg] = bininfo(sc,'ResStatus');
disp(bi)
```

```
         Bin            Good    Bad     Odds        WOE         InfoValue 
    ______________      ____    ___    ______    _________    __________

    {'Tenant'     }     296     161    1.8385    -0.095463     0.0035249
    {'Home Owner'}     352     171    2.0585     0.017549    0.00013382
    {'Other'      }     128      52    2.4615      0.19637     0.0055808
    {'<missing>'  }      27      13    2.0769     0.026469    2.3248e-05
    {'Totals'     }     803     397    2.0227          NaN     0.0092627
```

```
plotbins(sc,'ResStatus')
```



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 predictor 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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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 as the comma-separated pair consisting of `'BinText'` and 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 as the comma-separated pair consisting of `'WOE'` and 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 as the comma-separated pair consisting of `'Legend'` and 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.

Version History

Introduced in R2014b

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

`creditscorecard` | `autobinning` | `bininfo` | `predictorinfo` | `modifypredictor` | `bindata` | `modifybins` | `fitmodel` | `displaypoints` | `formatpoints` | `score` | `setmodel` | `probdefault` | `validatemodel`

Topics

“Case Study for Credit Scorecard Analysis” on page 8-70
“Credit Scorecard Modeling with Missing Values” on page 8-56
“Troubleshooting Credit Scorecard Results” on page 8-63
“Credit Scorecard Modeling Workflow” on page 8-51
“About Credit Scorecards” on page 8-47

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=46×6 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=7×6 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 = 5×1

       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=5×6 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=4×6 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=3×2 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=3×6 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=3×2 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=6×6 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=5×6 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=3×6 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: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIn
        NumericPredictors: {'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBalance'  'Uti
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
           BinMissingData: 0
                    IDVar: 'CustID'
            PredictorVars: {'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIncome'  'Tr
                     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=5×6 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=6×6 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=4×6 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=3×2 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=3×6 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=4×6 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=3×2 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=4×6 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=4×6 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=4×6 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
```

Modify Bin Information When Using Missing Data

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `dataMissing` with missing values.

```
load CreditCardData.mat
head(dataMissing,5)
```

```
    CustID    CustAge    TmAtAddress     ResStatus     EmpStatus    CustIncome    TmWBank    Othe
    ______    _______    ___________    ___________    _________    __________    _______    ____

      1          53          62         <undefined>    Unknown        50000         55         Ye
      2          61          22         Home Owner     Employed       52000         25         Ye
      3          47          30         Tenant         Employed       37000         61         No
      4         NaN          75         Home Owner     Employed       53000         20         Ye
      5          68          56         Home Owner     Employed       53000         14         Ye
```

```
fprintf('Number of rows: %d\n',height(dataMissing))
```

```
Number of rows: 1200
```

```
fprintf('Number of missing values CustAge: %d\n',sum(ismissing(dataMissing.CustAge)))
```

```
Number of missing values CustAge: 30
```

```
fprintf('Number of missing values ResStatus: %d\n',sum(ismissing(dataMissing.ResStatus)))
```

```
Number of missing values ResStatus: 40
```

Use `creditscorecard` with the name-value argument `'BinMissingData'` set to `true` to bin the missing data in a separate bin.

```
sc = creditscorecard(dataMissing,'IDVar','CustID','BinMissingData',true);
sc = autobinning(sc);

disp(sc)
```

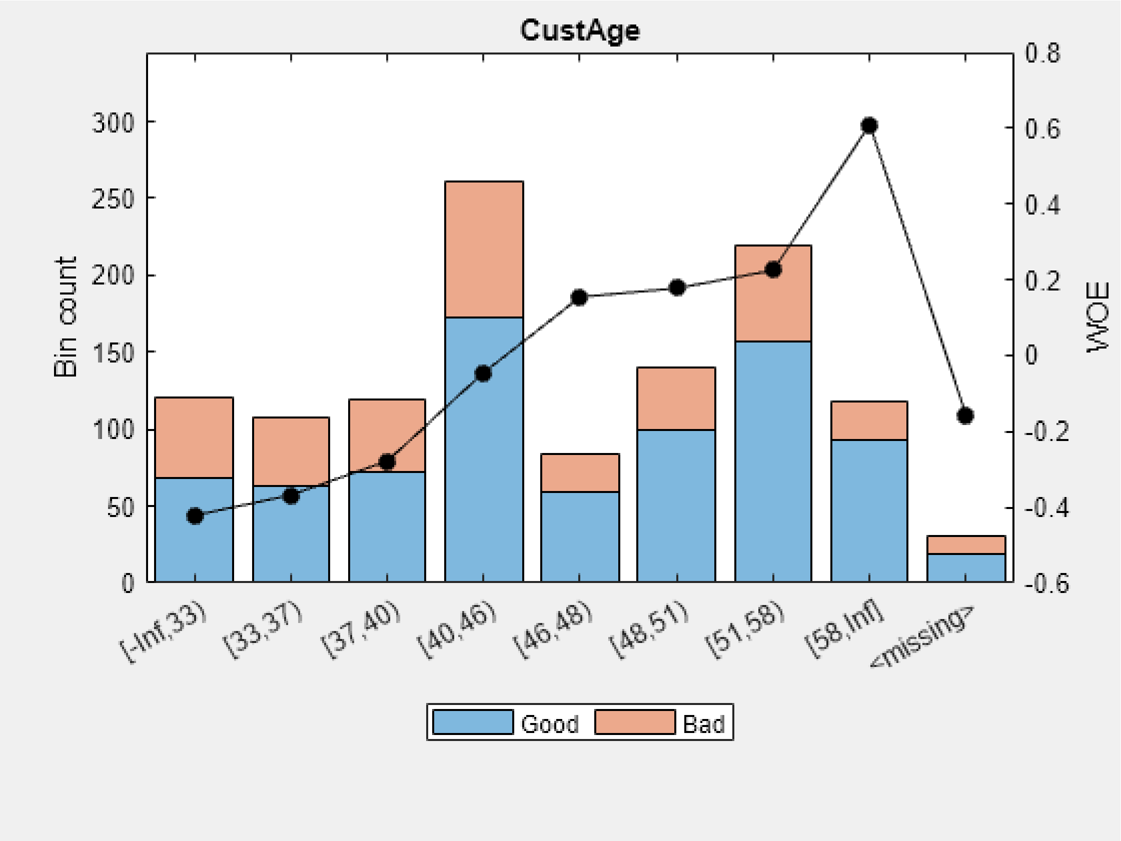
```
  creditscorecard with properties:

                GoodLabel: 0
              ResponseVar: 'status'
               WeightsVar: ''
                 VarNames: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIn
        NumericPredictors: {'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBalance'  'Util
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
           BinMissingData: 1
                    IDVar: 'CustID'
            PredictorVars: {'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIncome'  'Tm
                     Data: [1200x11 table]
```

Display bin information for numeric data for `'CustAge'` that includes missing data in a separate bin labelled `<missing>`.

```
[bi,cp] = bininfo(sc,'CustAge');
disp(bi)
```

```
         Bin         Good    Bad     Odds       WOE       InfoValue 
    _____________    ____    ___    ______    ________    __________

    {'[-Inf,33)'}      69     52    1.3269    -0.42156      0.018993
    {'[33,37)'  }      63     45       1.4    -0.36795      0.012839
    {'[37,40)'  }      72     47    1.5319     -0.2779     0.0079824
    {'[40,46)'  }     172     89    1.9326    -0.04556     0.0004549
    {'[46,48)'  }      59     25      2.36     0.15424     0.0016199
    {'[48,51)'  }      99     41    2.4146     0.17713     0.0035449
    {'[51,58)'  }     157     62    2.5323     0.22469     0.0088407
    {'[58,Inf]' }      93     25      3.72     0.60931      0.032198
    {'<missing>'}      19     11    1.7273    -0.15787    0.00063885
    {'Totals'   }     803    397    2.0227         NaN      0.087112
```

```
plotbins(sc,'CustAge')
```

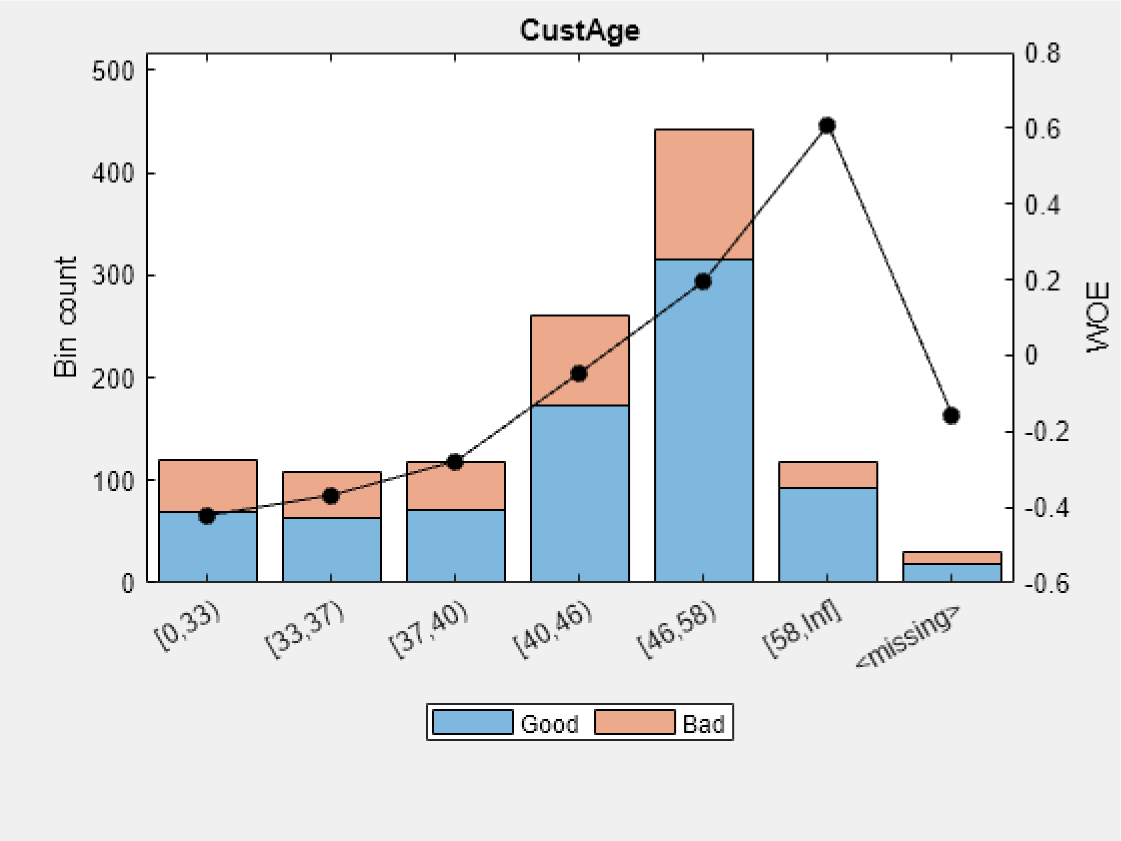


For the numeric predictor `CustAge`, remove cut points `48` and `51` and then use `modifybins` to define a `'MinValue'` of `0` to manually change the binning and notice that this does not affect the data in the `<missing>` bin and the `<missing>` bin remains at the end.

```
cp(cp==48) = [];
cp(cp==51) = [];
sc = modifybins(sc,'CustAge','CutPoints',cp,'MinValue',0);
bi = bininfo(sc,'CustAge');
disp(bi)
```

```
         Bin          Good    Bad     Odds        WOE       InfoValue 
    ______________    ____    ___    ______    ________    __________

    {'[0,33)'    }     69      52    1.3269    -0.42156      0.018993
    {'[33,37)'   }     63      45       1.4    -0.36795      0.012839
    {'[37,40)'   }     72      47    1.5319     -0.2779     0.0079824
    {'[40,46)'   }    172      89    1.9326    -0.04556     0.0004549
    {'[46,58)'   }    315     128    2.4609     0.19612      0.013701
    {'[58,Inf]'  }     93      25      3.72     0.60931      0.032198
    {'<missing>'}     19      11    1.7273    -0.15787    0.00063885
    {'Totals'    }    803     397    2.0227         NaN      0.086808
```

```
plotbins(sc,'CustAge')
```

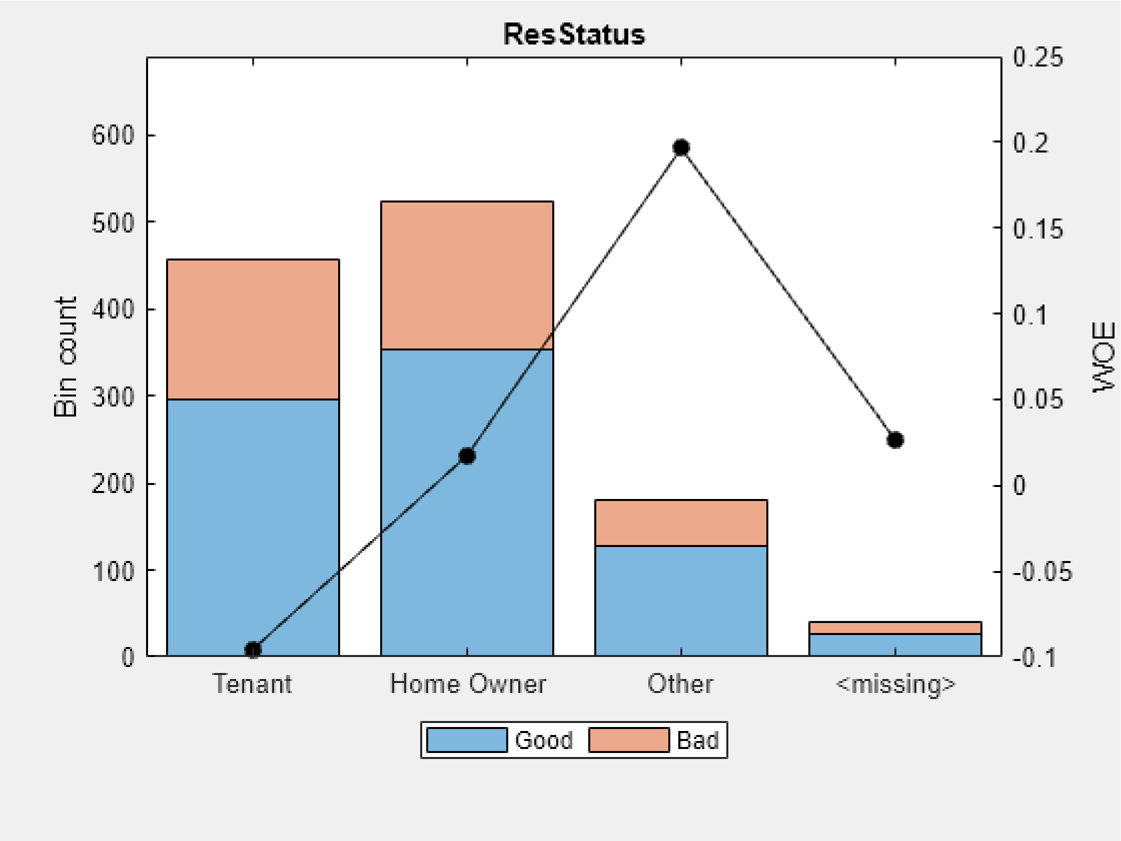


Display bin information for categorical data for `'ResStatus'` that includes missing data in a separate bin labelled `<missing>`.

```
[bi,cg] = bininfo(sc,'ResStatus');
disp(bi)
```

```
        Bin             Good    Bad     Odds         WOE       InfoValue 
    ________________    ____    ___    ______    _________    __________

    {'Tenant'     }     296     161    1.8385    -0.095463     0.0035249
    {'Home Owner'}      352     171    2.0585     0.017549    0.00013382
    {'Other'      }     128      52    2.4615      0.19637     0.0055808
    {'<missing>'  }      27      13    2.0769     0.026469    2.3248e-05
    {'Totals'     }     803     397    2.0227          NaN     0.0092627
```

```
plotbins(sc,'ResStatus')
```



For the categorical predictor `ResStatus`, use `modifybins` to manually merge `'HomeOwner'` and `'Other'` into a single group by assigning the same bin number to these categories. Notice that this does not affect the data in the `<missing>` bin and the `<missing>` bin remains at the end.

```
cg.BinNumber(3) = 2;
sc = modifybins(sc,'ResStatus','CatGrouping',cg);
[bi,cg] = bininfo(sc,'ResStatus');
disp(bi)
```

```
         Bin           Good    Bad     Odds         WOE        InfoValue 
    _____________      ____    ___    ______    _________    __________

    {'Group1'   }      296     161    1.8385    -0.095463     0.0035249
    {'Group2'   }      480     223    2.1525     0.062196     0.0022419
    {'<missing>'}       27      13    2.0769     0.026469    2.3248e-05
    {'Totals'   }      803     397    2.0227          NaN       0.00579
```

```
disp(cg)
```

```
       Category        BinNumber
    ______________     _________

    {'Tenant'    }         1    
    {'Home Owner'}         2    
    {'Other'     }         2    
```

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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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 the comma-separated pair consisting of `'MinValue'` and 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 the comma-separated pair consisting of `'MaxValue'` and 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 as the comma-separated pair consisting of `'CutPoints'` and a nondecreasing numeric array. If there are `NumBins` bins, there are *n* = `NumBins – 1` cut points so that C1, C2,..., C*n* 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 as the comma-separated pair consisting of `'CatGrouping'` and 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 as the comma-separated pair consisting of `'BinLabels'` and a cell array of character vectors with bin label names.

Note `'BinLabels'` does not support a value of `<missing>`.

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'`, and so on, for bin 1, bin 2, and so on, 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`.

Version History

Introduced in R2014b

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

`creditscorecard` | `autobinning` | `bininfo` | `predictorinfo` | `modifypredictor` | `bindata` | `plotbins` | `fitmodel` | `displaypoints` | `formatpoints` | `score` | `setmodel` | `probdefault` | `validatemodel`

Topics

“Case Study for Credit Scorecard Analysis” on page 8-70
“Credit Scorecard Modeling with Missing Values” on page 8-56
“Troubleshooting Credit Scorecard Results” on page 8-63
“Credit Scorecard Modeling Workflow” on page 8-51
“About Credit Scorecards” on page 8-47

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: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIn
        NumericPredictors: {'CustAge'  'TmAtAddress'  'ResStatus'  'CustIncome'  'TmWBank'  'AMBa
    CategoricalPredictors: {'EmpStatus'  'OtherCC'}
           BinMissingData: 0
                    IDVar: 'CustID'
            PredictorVars: {'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIncome'  'Tr
                     Data: [1200x11 table]
```

```
[T,Stats] = predictorinfo(sc,'ResStatus')
```

```
T=1×4 table
                 PredictorType     LatestBinning      LatestFillMissingType    LatestFillMissin
                 _____________    _________________    _____________________    ________________
```

```
    ResStatus     {'Numeric'}     {'Original Data'}       {'Original'}              {0x0 double
```

```
Stats=4×1 table
             Value 
            _______

    Min           1
    Max           3
    Mean     1.7017
    Std     0.71833
```

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: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIn
        NumericPredictors: {'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBalance'  'Uti
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
           BinMissingData: 0
                    IDVar: 'CustID'
            PredictorVars: {'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIncome'  'Tr
                     Data: [1200x11 table]
```

```
[T,Stats] = predictorinfo(sc,'ResStatus')
```

```
T=1×5 table
                 PredictorType     Ordinal      LatestBinning      LatestFillMissingType    Late
                _______________    _______    _________________    _____________________    ____

    ResStatus    {'Categorical'}     false     {'Original Data'}        {'Original'}
```

```
Stats=3×1 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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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 the comma-separated pair consisting of `'PredictorType'` and 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 the comma-separated pair consisting of `'Ordinal'` and 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.

Version History

Introduced in R2015b

See Also

`creditscorecard` | `modifybins` | `predictorinfo` | `bininfo`

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: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIn
        NumericPredictors: {'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBalance'  'Uti
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
           BinMissingData: 0
                    IDVar: 'CustID'
            PredictorVars: {'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIncome'  'Tr
                     Data: [1200x11 table]
```

Obtain the predictor statistics for the `PredictorName` of `CustAge`.

```
[T,Stats] = predictorinfo(sc,'CustAge')
```

```
T=1×4 table
               PredictorType      LatestBinning       LatestFillMissingType    LatestFillMissingVa
               _____________    __________________    _____________________    ___________________

    CustAge     {'Numeric'}     {'Original Data'}         {'Original'}              {0x0 double}

Stats=4×1 table
              Value
              _____
    Min         21
    Max         74
    Mean    45.174
    Std     9.8302
```

Obtain the predictor statistics for the `PredictorName` of `ResStatus`.

```
[T,Stats] = predictorinfo(sc,'ResStatus')
```

```
T=1×5 table
                  PredictorType     Ordinal      LatestBinning       LatestFillMissingType    Late
                 _______________    _______    _________________    _____________________    ____

    ResStatus    {'Categorical'}     false     {'Original Data'}        {'Original'}


Stats=3×1 table
                  Count
                  _____

    Home Owner     542
    Tenant         474
    Other          184
```

Obtain Information for a Specified `PredictorName` After Using `fillmissing`

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the data (using a dataset from Refaat 2011).

```
load CreditCardData
sc = creditscorecard(dataMissing,'BinMissingData',true,'IDVar','CustID');
sc = autobinning(sc);
```

Use `fillmissing` to replace missing values for the `CustAge` predictor with a value of `38`.

```
sc = fillmissing(sc,'CustAge','constant',38);
```

Obtain the predictor statistics for the `PredictorName` of `CustAge`.

```
[T,Stats] = predictorinfo(sc,'CustAge')
```

```
T=1×4 table
               PredictorType          LatestBinning           LatestFillMissingType    LatestFillMi
               _____________    ________________________    _____________________    ____________

    CustAge    {'Numeric'}      {'Automatic / Monotone'}        {'Constant'}                {[38


Stats=4×1 table
              Value
              ______

    Min         21
    Max         74
    Mean    44.932
    Std     9.7436
```

Use `fillmissing` to replace missing values for the `ResStatus` predictor with a `mode` value.

```
sc = fillmissing(sc,'ResStatus','mode');
```

Obtain the predictor statistics for the `PredictorName` of `ResStatus`.

```
[T,Stats] = predictorinfo(sc,'ResStatus')
```

```
T=1×5 table
                  PredictorType     Ordinal        LatestBinning           LatestFillMissingType
                 _______________    _______    ________________________    _____________________

    ResStatus    {'Categorical'}     false     {'Automatic / Monotone'}          {'Mode'}

Stats=3×1 table
                  Count
                  _____

    Tenant         457
    Home Owner     563
    Other          180
```

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.
- `'LatestFillMissingType'` — If `fillmissing` has been applied to the predictor, the value of the `Statistics` argument for `fillmissing` is displayed. If the predictor does not have any missing data, then the fill type is `'Original'`.
- `'LatestFillMissingValue'` — If `fillmissing` has been applied to the predictor, the fill value is displayed. If the predictor does not have any missing data, then the fill value is `[ ]`.

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.

Version History

Introduced in R2015b

See Also

`creditscorecard` | `modifybins` | `modifypredictor` | `bininfo` | `fillmissing`

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=4×6 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
```

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: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIn
        NumericPredictors: {'CustID'  'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBalan
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
           BinMissingData: 0
                    IDVar: ''
            PredictorVars: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIn
                     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=10×6 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. The Weight-of-Evidence (WOE) is defined bin by bin, but there is no concept of "total WOE", therefore the last element in the `'Totals'` row is set to `NaN`.

```
bi = bininfo(sc,'ResStatus','Statistics','WOE');
disp(bi)
```

```
         Bin           Good    Bad       WOE    
    ______________     ____    ___    _________

    {'Home Owner'}     365     177     0.019329
    {'Tenant'    }     307     167    -0.095564
    {'Other'     }     131      53      0.20049
    {'Totals'    }     803     397          NaN
```

Display customized bin information for the categorical predictor `ResStatus`, keeping only the `Odds` and `WOE` columns, without the `Totals` row.

```
bi = bininfo(sc,'ResStatus','Statistics',{'Odds','WOE'},'Totals','Off');
disp(bi)
```

```
         Bin           Good    Bad     Odds        WOE    
    ______________     ____    ___    ______    _________

    {'Home Owner'}     365     177    2.0621     0.019329
    {'Tenant'    }     307     167    1.8383    -0.095564
    {'Other'     }     131      53    2.4717      0.20049
```

Display information value, entropy, Gini, and chi square statistics. For more information on these statistics, see “Statistics for a Credit Scorecard” on page 15-1806.

For information value, entropy and Gini, the value reported at a bin level is the contribution of the bin to the total value. The total information value, entropy, and Gini measures are in the `'Totals'` row.

For chi square, if there are *N* bins, the first *N-1* values in the `'Chi2'` column report pairwise chi square statistics for contiguous bins. For example, this pairwise measure is also used by the `'Merge'` algorithm in `autobinning` to determine if two contiguous bins should be merged. In this example, the first value in the `'Chi2'` column (`1.0331`) is the chi square statistic of bins 1 and 2 (`'Home Owner'` and `'Tenant'`), and the second value in the column (`2.5103`) is the chi square statistic of bins 2 and 3 (`'Tenant'` and `'Other'`). There are no more pairwise chi square values to compute in this example, so the third element of the `'Chi2'` column is set to `NaN`. The chi square value reported in the `'Totals'` row is the chi square statistic computed over all bins.

```
bi = bininfo(sc,'ResStatus','Statistics',{'InfoValue','Entropy','Gini','Chi2'});
disp(bi)
```

```
         Bin           Good    Bad    InfoValue    Entropy      Gini       Chi2 
    ______________     ____    ___    _________    _______    _______    ______
    {'Home Owner'}    365     177    0.0001682    0.91138    0.43984    1.0331
    {'Tenant'    }    307     167    0.0036638    0.93612    0.45638    2.5103
    {'Other'     }    131      53    0.0059418    0.86618    0.41015       NaN
    {'Totals'    }    803     397    0.0097738    0.91422    0.44182    2.5549
```

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=4×6 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=3×2 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=3×6 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=3×2 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=46×6 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=8×6 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 = 6×1

       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=7×6 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 = 5×1

       29000
       35000
       40000
       42000
       47000

mv = 1×2
```

```
    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.

Display Bin Information for Missing Data

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `dataMissing` with missing values.

```
load CreditCardData.mat
head(dataMissing,5)
```

```
    CustID    CustAge    TmAtAddress     ResStatus     EmpStatus    CustIncome    TmWBank    Othe
    ______    _______    ___________    ___________    _________    __________    _______    ____

      1          53          62         <undefined>    Unknown        50000         55         Ye
      2          61          22         Home Owner     Employed       52000         25         Ye
      3          47          30         Tenant         Employed       37000         61         No
      4         NaN          75         Home Owner     Employed       53000         20         Ye
      5          68          56         Home Owner     Employed       53000         14         Ye
```

```
fprintf('Number of rows: %d\n',height(dataMissing))
```

```
Number of rows: 1200
```

```
fprintf('Number of missing values CustAge: %d\n',sum(ismissing(dataMissing.CustAge)))
```

```
Number of missing values CustAge: 30
```

```
fprintf('Number of missing values ResStatus: %d\n',sum(ismissing(dataMissing.ResStatus)))
```

```
Number of missing values ResStatus: 40
```

Use `creditscorecard` with the name-value argument `'BinMissingData'` set to `true` to bin the missing data in a separate bin.

```
sc = creditscorecard(dataMissing,'IDVar','CustID','BinMissingData',true);
sc = autobinning(sc);

disp(sc)
```

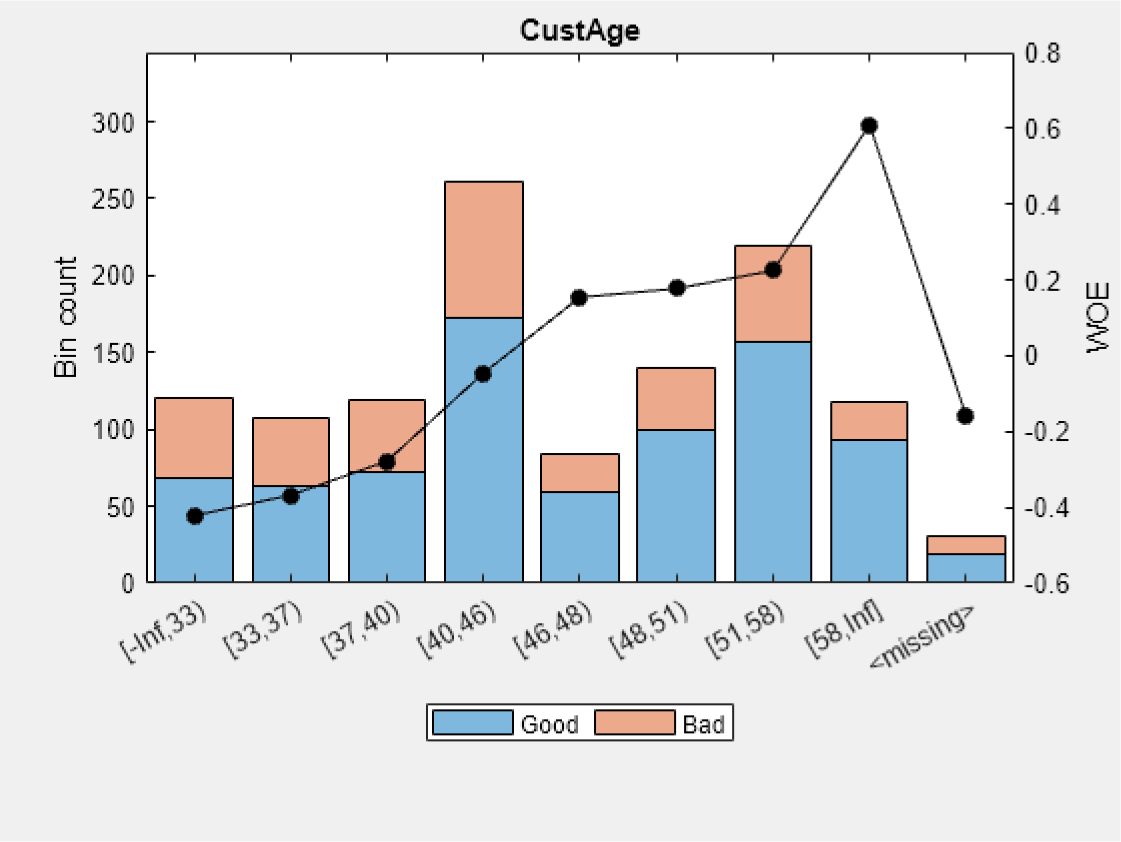
```
  creditscorecard with properties:

              GoodLabel: 0
            ResponseVar: 'status'
             WeightsVar: ''
               VarNames: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIn
      NumericPredictors: {'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBalance'  'Uti
  CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
         BinMissingData: 1
                  IDVar: 'CustID'
          PredictorVars: {'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIncome'  'Tr
                   Data: [1200x11 table]
```

Display bin information for numeric data for `'CustAge'` that includes missing data in a separate bin labelled `<missing>`.

```
bi = bininfo(sc,'CustAge');
disp(bi)
```

```
         Bin           Good    Bad     Odds        WOE       InfoValue 
    ______________     ____    ___    ______    ________    __________

    {'[-Inf,33)'}       69     52     1.3269    -0.42156      0.018993
    {'[33,37)'  }       63     45        1.4    -0.36795      0.012839
    {'[37,40)'  }       72     47     1.5319     -0.2779     0.0079824
    {'[40,46)'  }      172     89     1.9326    -0.04556     0.0004549
    {'[46,48)'  }       59     25       2.36     0.15424     0.0016199
    {'[48,51)'  }       99     41     2.4146     0.17713     0.0035449
    {'[51,58)'  }      157     62     2.5323     0.22469     0.0088407
    {'[58,Inf]' }       93     25       3.72     0.60931      0.032198
    {'<missing>'}       19     11     1.7273    -0.15787    0.00063885
    {'Totals'   }      803    397     2.0227         NaN      0.087112
```

```
plotbins(sc,'CustAge')
```



Display bin information for categorical data for `'ResStatus'` that includes missing data in a separate bin labelled `<missing>`.

```
[bi,cg] = bininfo(sc,'ResStatus');
disp(bi)
```

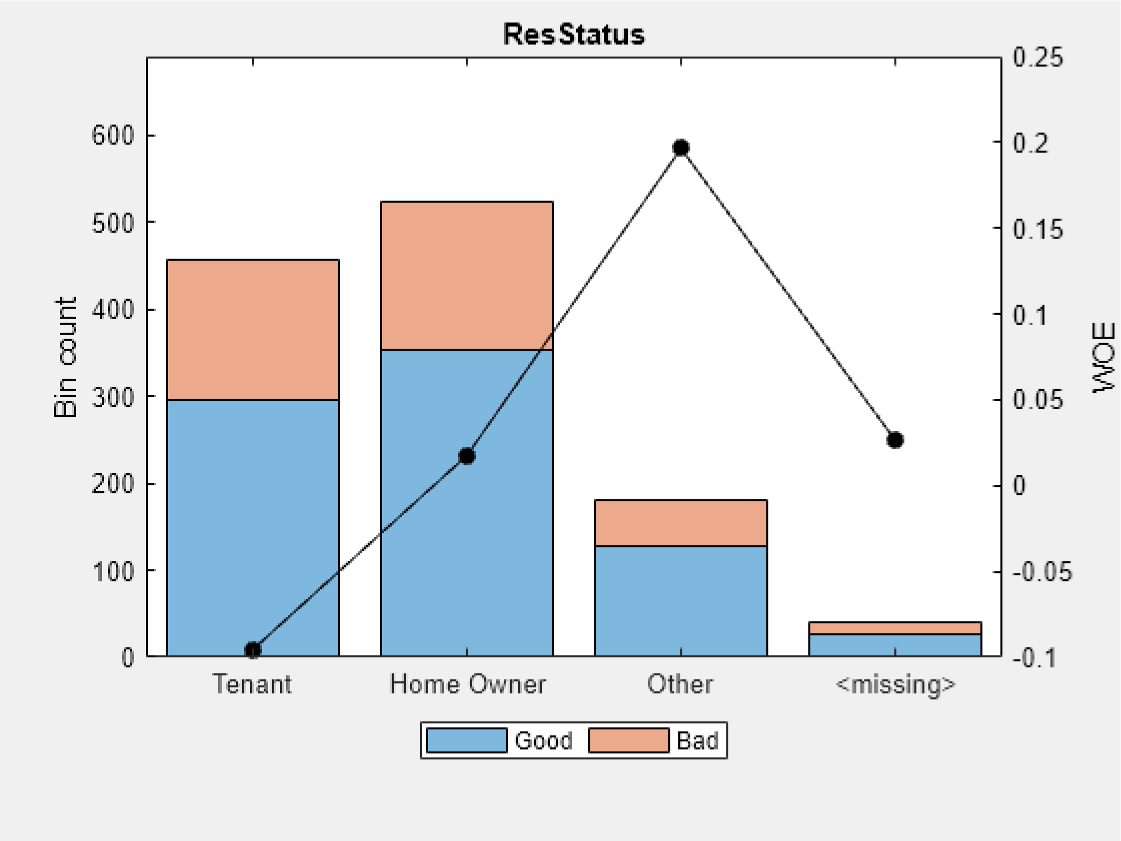
```
         Bin            Good    Bad     Odds        WOE        InfoValue 
    ______________      ____    ___    ______    _________    __________

    {'Tenant'     }     296     161    1.8385    -0.095463     0.0035249
    {'Home Owner'}      352     171    2.0585     0.017549    0.00013382
    {'Other'      }     128      52    2.4615      0.19637     0.0055808
    {'<missing>'  }      27      13    2.0769     0.026469    2.3248e-05
    {'Totals'     }     803     397    2.0227          NaN     0.0092627
```

```
disp(cg)
```

```
       Category         BinNumber
    ______________      _________

    {'Tenant'     }         1    
    {'Home Owner'}          2    
    {'Other'      }         3    
```

Note that the category grouping table does not include `<missing>` because this is a reserved bin and users cannot interact directly with the `<missing>` bin.

```
plotbins(sc,'ResStatus')
```



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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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'`, `'Gini'`, `'Chi2'` | cell array of character vectors with values `'Odds'`, `'WOE'`, `'InfoValue'`, `'Entropy'`,`'Gini'`, `'Chi2'`

List of statistics to include in the bin information, specified as the comma-separated pair consisting of `'Statistics'` and a character vector or a cell array of character vectors. For more information, see “Statistics for a Credit Scorecard” on page 15-1806. 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.
- `'Gini'` — Measure of statistical dispersion or inequality within a sample of data.
- `'Chi2'` — Measure of statistical difference and independence between groups.

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 the comma-separated pair consisting of `'Totals'` and 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.

More About

Statistics for a Credit Scorecard

Weight of Evidence (WOE) is a measure of the difference of the distribution of “Goods” and “Bads” within a bin.

Suppose the predictor's data takes on *M* possible values 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 b*`i`* 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 `InfoValuel`(*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.
```

Chi2 is computed pairwise for each pair of bins and measures the statistical difference between two groups when splitting or merging bins and is defined as:

```
 Chi2 = sum(sum((Aij - Eij)^2/Eij , j=1..k), i=m,m+1).
```

For more information on splitting and merging bins, see Split on page 15-1830 and Merge on page 15-1833.

Gini ratio is a measure of the parent node, that is, of the given bins/categories prior to splitting or merging. The Gini ratio is defined as:

```
Gr = 1-G_hat/Gp
```

`G_hat` is the weighted Gini measure for the current split or merge:

```
G_hat = Sum((nj/N) * Gj, j=1..m).
```

For more information on splitting and merging bins, see Split on page 15-1830 and Merge on page 15-1833.

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 8-54.

Version History

Introduced in R2014b

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

`creditscorecard` | `autobinning` | `predictorinfo` | `modifypredictor` | `modifybins` | `bindata` | `plotbins` | `fitmodel` | `displaypoints` | `formatpoints` | `score` | `setmodel` | `probdefault` | `validatemodel`

Topics

“Case Study for Credit Scorecard Analysis” on page 8-70
“Credit Scorecard Modeling with Missing Values” on page 8-56
“Troubleshooting Credit Scorecard Results” on page 8-63
“Credit Scorecard Modeling Workflow” on page 8-51
“About Credit Scorecards” on page 8-47
“Credit Scorecard Modeling Using Observation Weights” on page 8-54

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);
```

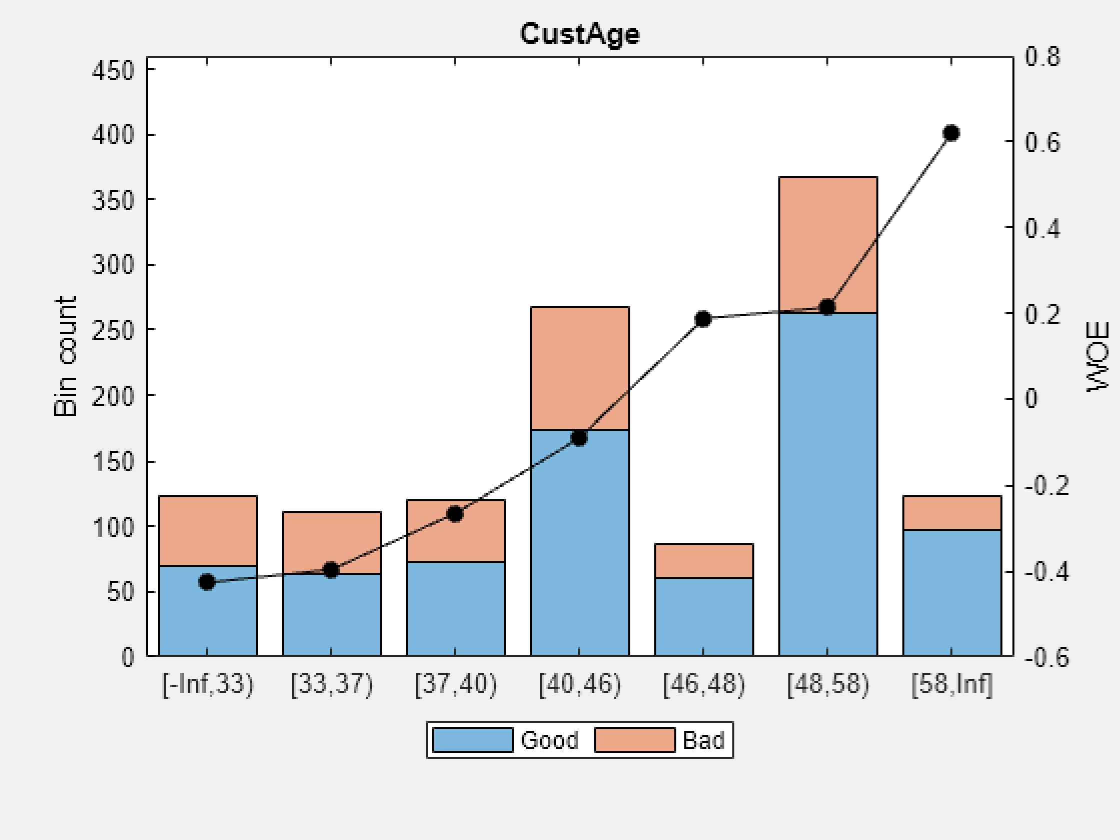
Use `bininfo` to display the binned data for the predictor `CustAge`.

```
bi = bininfo(sc, 'CustAge')
```

```
bi=8×6 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
```

Use `plotbins` to display the histogram and WOE curve for the predictor `CustAge`.

```
plotbins(sc,'CustAge')
```



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=8×6 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=11×6 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 = 9×1

       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=11×6 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=11×6 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=8×6 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=4×6 table
         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;
```

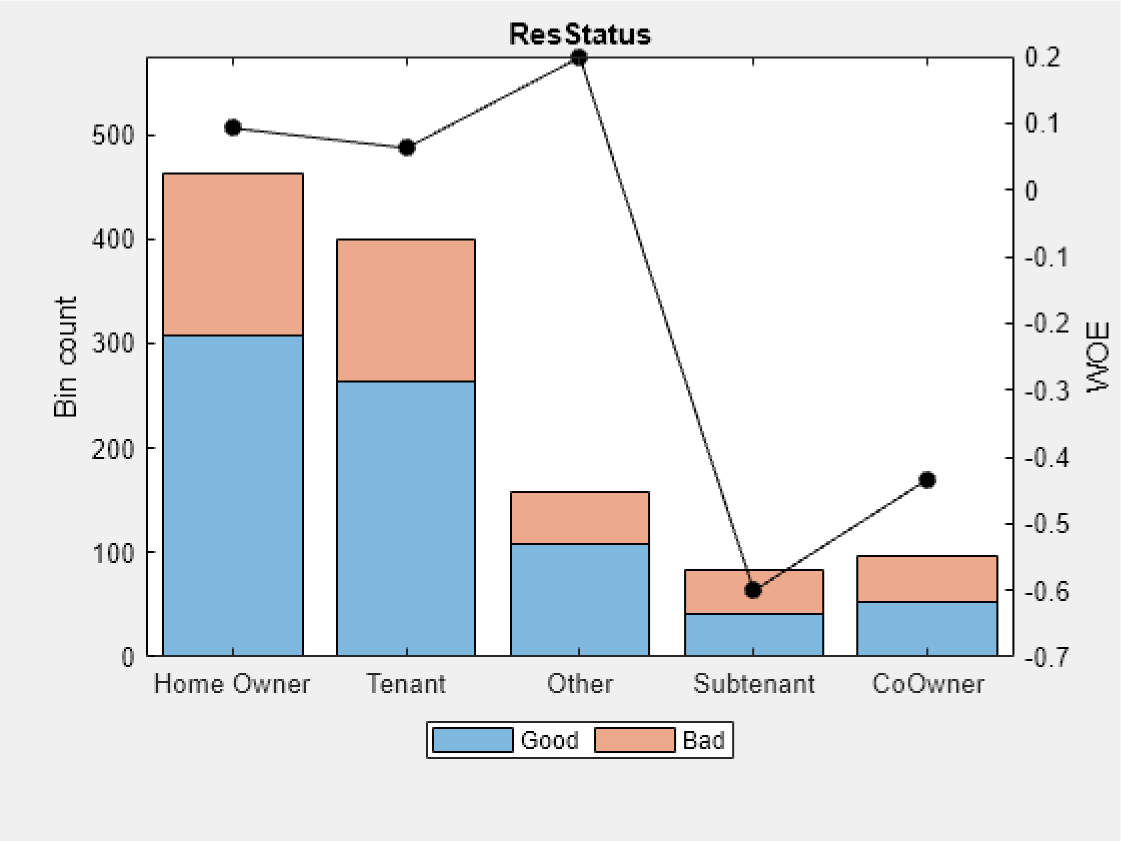
Update the `creditscorecard` object using the `newdata` and plot the bins for a later comparison.

```
scnew = creditscorecard(newdata,'IDVar','CustID');
[bi,cg] = bininfo(scnew,Predictor)
```

```
bi=6×6 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=5×2 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);
```

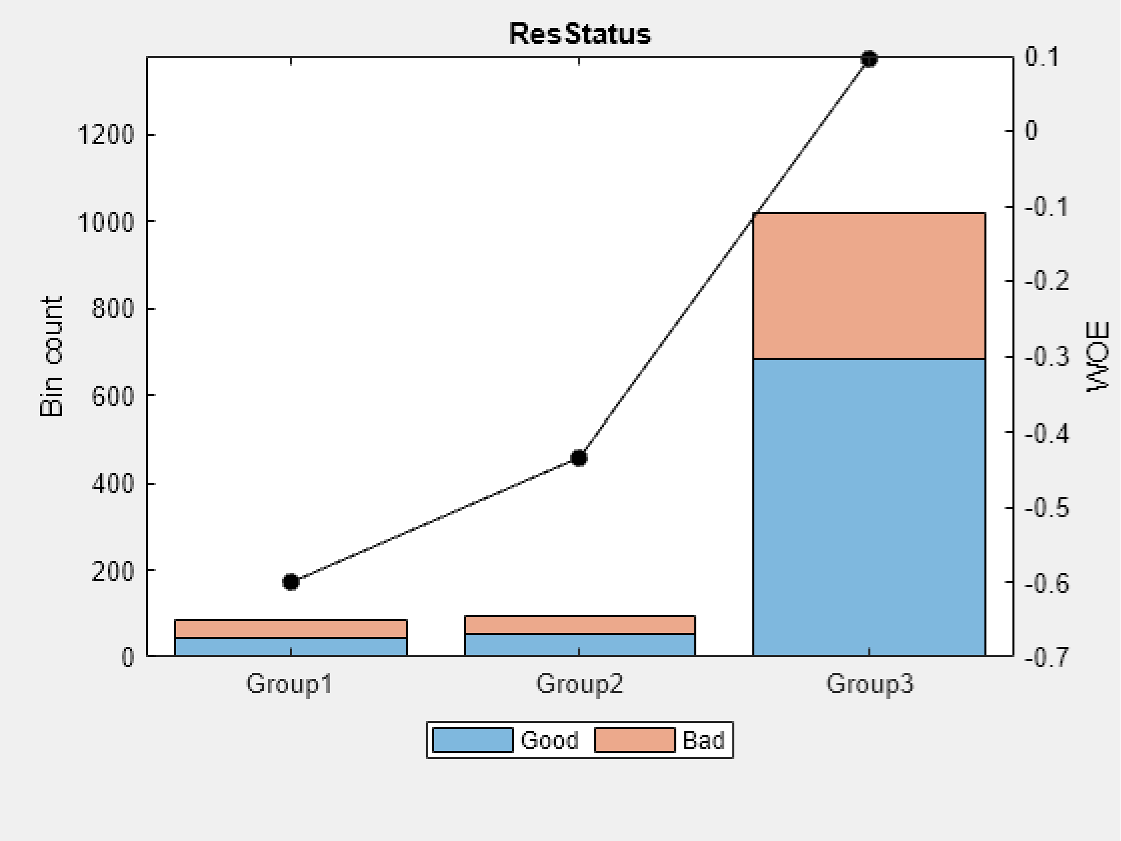
Use `bininfo` to display the bin information. The second output parameter `'cg'` captures the bin membership, which is the bin number that each group belongs to.

```
[bi,cg] = bininfo(scnew,Predictor)
```

```
bi=4×6 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=5×2 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)
```



Perform Automatic Binning When Using Missing Data

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `dataMissing` with missing values.

```
load CreditCardData.mat
head(dataMissing,5)
```

```
    CustID    CustAge    TmAtAddress     ResStatus     EmpStatus    CustIncome    TmWBank    Othe
    ______    _______    ___________    ___________    _________    __________    _______    ____

      1          53          62         <undefined>    Unknown        50000         55         Ye
      2          61          22         Home Owner     Employed       52000         25         Ye
      3          47          30         Tenant         Employed       37000         61         No
      4         NaN          75         Home Owner     Employed       53000         20         Ye
      5          68          56         Home Owner     Employed       53000         14         Ye
```

```
fprintf('Number of rows: %d\n',height(dataMissing))
```

```
Number of rows: 1200
```

```
fprintf('Number of missing values CustAge: %d\n',sum(ismissing(dataMissing.CustAge)))
```

```
Number of missing values CustAge: 30
```

```
fprintf('Number of missing values ResStatus: %d\n',sum(ismissing(dataMissing.ResStatus)))
```

```
Number of missing values ResStatus: 40
```

Use `creditscorecard` with the name-value argument `'BinMissingData'` set to `true` to bin the missing numeric and categorical data in a separate bin.

```
sc = creditscorecard(dataMissing,'BinMissingData',true);
disp(sc)
```

```
  creditscorecard with properties:

                GoodLabel: 0
              ResponseVar: 'status'
               WeightsVar: ''
                 VarNames: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIn
        NumericPredictors: {'CustID'  'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBalar
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
           BinMissingData: 1
                    IDVar: ''
            PredictorVars: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIn
                     Data: [1200x11 table]
```

Perform automatic binning using the `Merge` algorithm.

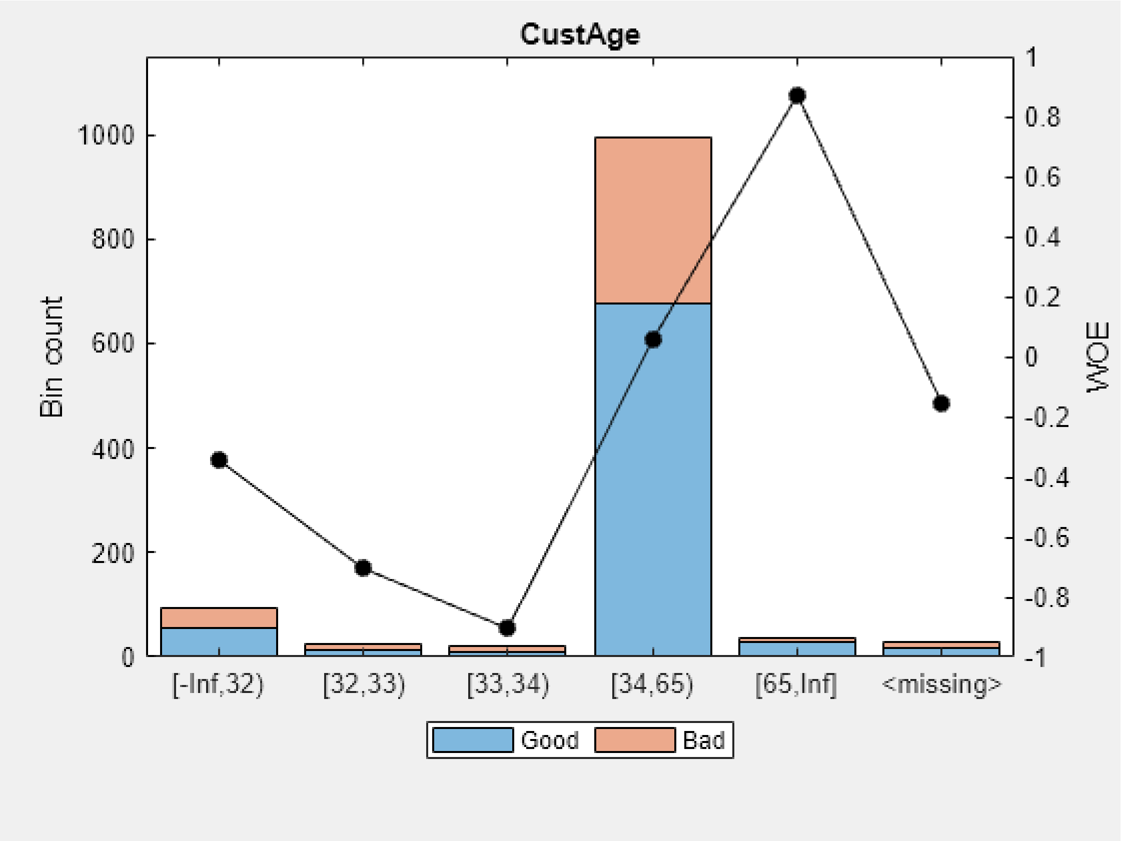
```
sc = autobinning(sc,'Algorithm','Merge');
```

Display bin information for numeric data for `'CustAge'` that includes missing data in a separate bin labelled `<missing>` and this is the last bin. No matter what binning algorithm is used in `autobinning`, the algorithm operates on the non-missing data and the bin for the `<missing>` numeric values for a predictor is always the last bin.

```
[bi,cp] = bininfo(sc,'CustAge');
disp(bi)
```

```
         Bin         Good    Bad     Odds        WOE       InfoValue 
    _____________    ____    ___    _______    ________    __________

    {'[-Inf,32)'}     56      39     1.4359    -0.34263     0.0097643
    {'[32,33)'  }     13      13          1    -0.70442      0.011663
    {'[33,34)'  }      9      11    0.81818    -0.90509      0.014934
    {'[34,65)'  }    677     317     2.1356    0.054351      0.002424
    {'[65,Inf]' }     29       6     4.8333     0.87112      0.018295
    {'<missing>'}     19      11     1.7273    -0.15787    0.00063885
    {'Totals'   }    803     397     2.0227         NaN      0.057718
```

```
plotbins(sc,'CustAge')
```

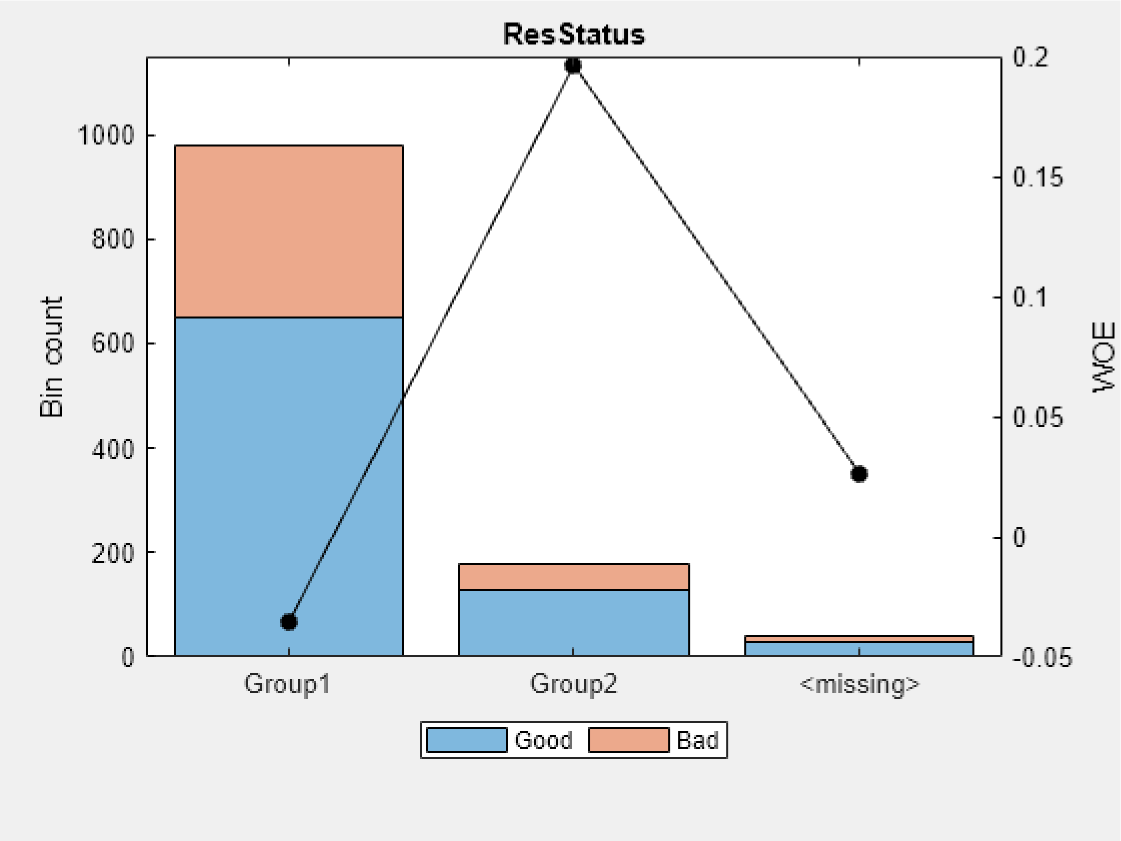


Display bin information for categorical data for `'ResStatus'` that includes missing data in a separate bin labelled `<missing>` and this is the last bin. No matter what binning algorithm is used in `autobinning`, the algorithm operates on the non-missing data and the bin for the `<missing>` categorical values for a predictor is always the last bin.

```
[bi,cg] = bininfo(sc,'ResStatus');
disp(bi)
```

```
         Bin           Good    Bad     Odds         WOE         InfoValue 
    ______________     ____    ___    ______    _________    __________

    {'Group1'     }    648     332    1.9518    -0.035663     0.0010449
    {'Group2'     }    128      52    2.4615      0.19637     0.0055808
    {'<missing>'}       27      13    2.0769     0.026469    2.3248e-05
    {'Totals'     }    803     397    2.0227          NaN     0.0066489
```

```
plotbins(sc,'ResStatus')
```



Perform Automatic Binning Using the Split Algorithm

This example demonstrates using the `'Split'` algorithm with categorical and numeric predictors. Load the `CreditCardData.mat` dataset and modify so that it contains four categories for the predictor '`ResStatus'` to demonstrate how the split algorithm works.

```
load CreditCardData.mat
x = data.ResStatus;
Ind = find(x == 'Tenant');
Nx  = length(Ind);
x(Ind(1:floor(Nx/3))) = 'Subletter';
data.ResStatus = x;
```

Create a `creditscorecard` and use `bininfo` to display the `'Statistics'`.

```
sc = creditscorecard(data,'IDVar','CustID');
[bi1,cg1] = bininfo(sc,'ResStatus','Statistics',{'Odds','WOE','InfoValue'});
disp(bi1)
```

```
        Bin            Good    Bad     Odds        WOE        InfoValue
    ______________     ____    ___    ______    _________    _________

    {'Home Owner'}     365     177    2.0621     0.019329    0.0001682
    {'Tenant'    }     204     112    1.8214      -0.1048    0.0029415
    {'Other'     }     131      53    2.4717      0.20049    0.0059418
    {'Subletter' }    103      55     1.8727    -0.077023    0.00079103
    {'Totals'    }    803     397     2.0227          NaN     0.0098426
```

```
disp(cg1)
```

```
       Category       BinNumber
    ______________    _________

    {'Home Owner'}        1    
    {'Tenant'    }        2    
    {'Other'     }        3    
    {'Subletter' }        4    
```

Using the Split Algorithm with a Categorical Predictor

Apply presorting to the `'ResStatus'` category using the default sorting by `'Odds'` and specify the `'Split'` algorithm.

```
sc = autobinning(sc,'ResStatus', 'Algorithm', 'split','AlgorithmOptions',...
    {'Measure','gini','SortCategories','odds','Tolerance',1e4});
[bi2,cg2] = bininfo(sc,'ResStatus','Statistics',{'Odds','WOE','InfoValue'});
disp(bi2)
```

```
       Bin        Good    Bad     Odds     WOE    InfoValue
    __________    ____    ___    ______    ___    _________

    {'Group1'}    803     397    2.0227      0        0    
    {'Totals'}    803     397    2.0227    NaN        0    
```

```
disp(cg2)
```

```
       Category       BinNumber
    ______________    _________

    {'Tenant'    }        1    
    {'Subletter' }        1    
    {'Home Owner'}        1    
    {'Other'     }        1    
```

Using the Split Algorithm with a Numeric Predictor

To demonstrate a split for the numeric predictor, 'TmAtAddress', first use `autobinning` with the default `'Monotone'` algorithm.

```
sc = autobinning(sc,'TmAtAddress');
bi3 = bininfo(sc,'TmAtAddress','Statistics',{'Odds','WOE','InfoValue'});
disp(bi3)
```

```
         Bin          Good    Bad     Odds        WOE        InfoValue 
    _____________    ____    ___    ______    _________    __________

    {'[-Inf,23)'}    239     129    1.8527    -0.087767     0.0023963
    {'[23,83)'  }    480     232     2.069      0.02263    0.00030269
    {'[83,Inf]' }     84      36    2.3333      0.14288       0.00199
    {'Totals'   }    803     397    2.0227          NaN      0.004689
```

Then use `autobinning` with the `'Split'` algorithm.

```
sc = autobinning(sc,'TmAtAddress','Algorithm', 'Split');
bi4 = bininfo(sc,'TmAtAddress','Statistics',{'Odds','WOE','InfoValue'});
disp(bi4)
```

```
        Bin          Good    Bad      Odds         WOE         InfoValue 
    ____________     ____    ___    _______    _________    __________

    {'[-Inf,4)'}      20      12     1.6667     -0.19359     0.0010299
    {'[4,5)'   }       4       7    0.57143       -1.264      0.015991
    {'[5,23)'  }     215     110     1.9545    -0.034261    0.00031973
    {'[23,33)' }     130      39     3.3333      0.49955        0.0318
    {'[33,Inf]'}     434     229     1.8952    -0.065096     0.0023664
    {'Totals'  }     803     397     2.0227          NaN      0.051507
```

Perform Automatic Binning Using the Merge Algorithm

Load the `CreditCardData.mat` dataset. This example demonstrates using the `'Merge'` algorithm with categorical and numeric predictors.

```
load CreditCardData.mat
```

Using the Merge Algorithm with a Categorical Predictor

To merge a categorical predictor, create a `creditscorecard` using default sorting by `'Odds'` and then use `bininfo` on the categorical predictor `'ResStatus'`.

```
sc = creditscorecard(data,'IDVar','CustID');
[bi1,cg1] = bininfo(sc,'ResStatus','Statistics',{'Odds','WOE','InfoValue'});
disp(bi1);
```

```
         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
```

```
disp(cg1);
```

```
       Category        BinNumber
    ______________     _________

    {'Home Owner'}         1    
    {'Tenant'    }         2    
    {'Other'     }         3    
```

Use `autobinning` and specify the `'Merge'` algorithm.

```
sc = autobinning(sc,'ResStatus','Algorithm', 'Merge');
[bi2,cg2] = bininfo(sc,'ResStatus','Statistics',{'Odds','WOE','InfoValue'});
disp(bi2)
```

```
       Bin         Good    Bad     Odds        WOE        InfoValue
    __________     ____    ___    ______    _________    _________

    {'Group1'}     672     344    1.9535    -0.034802    0.0010314
    {'Group2'}    131      53    2.4717      0.20049    0.0059418
    {'Totals'}    803     397    2.0227          NaN    0.0069732
```

```
disp(cg2)
```

```
       Category       BinNumber
    ______________    _________

    {'Tenant'    }        1    
    {'Home Owner'}        1    
    {'Other'     }        2    
```

Using the Merge Algorithm with a Numeric Predictor

To demonstrate a merge for the numeric predictor, 'TmAtAddress', first use `autobinning` with the default `'Monotone'` algorithm.

```
sc = autobinning(sc,'TmAtAddress');
bi3 = bininfo(sc,'TmAtAddress','Statistics',{'Odds','WOE','InfoValue'});
disp(bi3)
```

```
         Bin         Good    Bad     Odds        WOE        InfoValue 
    _____________    ____    ___    ______    _________    __________

    {'[-Inf,23)'}    239     129    1.8527    -0.087767     0.0023963
    {'[23,83)'  }    480     232     2.069      0.02263    0.00030269
    {'[83,Inf]' }     84      36    2.3333      0.14288       0.00199
    {'Totals'   }    803     397    2.0227          NaN      0.004689
```

Then use `autobinning` with the `'Merge'` algorithm.

```
sc = autobinning(sc,'TmAtAddress','Algorithm', 'Merge');
bi4 = bininfo(sc,'TmAtAddress','Statistics',{'Odds','WOE','InfoValue'});
disp(bi4)
```

```
         Bin         Good    Bad     Odds         WOE        InfoValue 
    _____________    ____    ___    _______    _________    __________

    {'[-Inf,28)'}    303     152     1.9934    -0.014566    8.0646e-05
    {'[28,30)'  }     27       2       13.5       1.8983      0.054264
    {'[30,98)'  }    428     216     1.9815    -0.020574    0.00022794
    {'[98,106)' }     11      13    0.84615     -0.87147      0.016599
    {'[106,Inf]'}     34      14     2.4286      0.18288     0.0012942
    {'Totals'   }    803     397     2.0227          NaN      0.072466
```

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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `sc = autobinning(sc,'Algorithm','EqualFrequency')`

`Algorithm` — Algorithm selection

`'Monotone'` (default) | character vector with values `'Monotone'`, `'Split'`, `'Merge'`, `'EqualFrequency'`, `'EqualWidth'`

Algorithm selection, specified as the comma-separated pair consisting of `'Algorithm'` and 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 15-1828.
- `'Split'` — Supervised binning algorithm, where a measure is used to split the data into bins. The measures supported by `'Split'` are `gini`, `chi2`, `infovalue`, and `entropy`. The resulting split must be such that the gain in the information function is maximized. For more information on these measures, see `AlgorithmOptions` and “Split” on page 15-1830.
- `'Merge'` — Supervised automatic binning algorithm, where a measure is used to merge bins into buckets. The measures supported by `'Merge'` are `chi2`, `gini`, `infovalue`, and `entropy`. The resulting merging must be such that any pair of adjacent bins is statistically different from each other, according to the chosen measure. For more information on these measures, see `AlgorithmOptions` and “Merge” on page 15-1833.
- `'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 15-1835.
- `'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 15-1836.

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 as the comma-separated pair consisting of 'AlgorithmOptions' and 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 15-1828.
  - {'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 15-1836
- For Split algorithm:
  - {'InitialNumBins',*n*} — Specifies an integer that determines the number (*n* >0) of bins that the predictor is initially binned into before splitting. Valid for numeric predictors only. Default is 50.
  - {'Measure',*MeasureName*} — Specifies the measure where '*MeasureName*' is one of the following:'Gini' (default), 'Chi2', 'InfoValue', or 'Entropy'.
  - {'MinBad',*n*} — Specifies the minimum number *n* (*n*>=0) of Bads per bin. The default value is 1, to avoid pure bins.
  - {'MaxBad',*n*} — Specifies the maximum number *n* (*n*>=0) of Bads per bin. The default value is Inf.

  - `{'MinGood',n}` — Specifies the minimum number *n* (*n*`>=0`) of Goods per bin. The default value is `1`, to avoid pure bins.
  - `{'MaxGood',n}` — Specifies the maximum number *n* (*n*`>=0`) of Goods per bin. The default value is `Inf`.
  - `{'MinCount',n}` — Specifies the minimum number *n* (*n*`>=0`) of observations per bin. The default value is `1`, to avoid empty bins.
  - `{'MaxCount',n}` — Specifies the maximum number *n* (*n*`>=0`) of observations per bin. The default value is `Inf`.
  - `{'MaxNumBins',n}` — Specifies the maximum number *n* (*n*`>=2`) of bins resulting from the splitting. The default value is `5`.
  - `{'Tolerance',Tol}` — Specifies the minimum gain (>0) in the information function, during the iteration scheme, to select the cut-point that maximizes the gain. The default is `1e4`.
  - `{'Significance',n}` — Significance level threshold for the chi-square statistic, above which splitting happens. Values are in the interval `[0,1]`. Default is `0.9` (90% significance level).
  - `{'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:

    - `'Goods'` — The categories are sorted by order of increasing values of “Good.”
    - `'Bads'` — The categories are sorted by order of increasing values of “Bad.”
    - `'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.
    - `'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 15-1836

- For `Merge` algorithm:

  - `{'InitialNumBins',n}` — Specifies an integer that determines the number (*n* >0) of bins that the predictor is initially binned into before merging. Valid for numeric predictors only. Default is `50`.
  - `{'Measure',MeasureName}` — Specifies the measure where '*MeasureName*' is one of the following:`'Chi2'` (default), `'Gini'`, `'InfoValue'`, or `'Entropy'`.
  - `{'MinNumBins',n}` — Specifies the minimum number *n* (*n*`>=2`) of bins that result from merging. The default value is `2`.
  - `{'MaxNumBins',n}` — Specifies the maximum number *n* (*n*`>=2`) of bins that result from merging. The default value is `5`.
  - `{'Tolerance',n}` — Specifies the minimum threshold below which merging happens for the information value and entropy statistics. Valid values are in the interval `(0.1)`. Default is `1e3`.
  - `{'Significance',n}` — Significance level threshold for the chi-square statistic, below which merging happens. Values are in the interval `[0,1]`. Default is `0.9` (90% significance level).
  - `{'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:

    - `'Goods'` — The categories are sorted by order of increasing values of “Good.”
    - `'Bads'` — The categories are sorted by order of increasing values of “Bad.”
    - `'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.
    - `'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 15-1836
- 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 15-1836
- 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 15-1836

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 as the comma-separated pair consisting of `'Display'` and 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.

More About

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 15-1837.

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 15-1836 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		
'[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
'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 15-1836 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'`).

Split

Split is a supervised automatic binning algorithm, where a measure is used to split the data into buckets. The supported measures are `gini`, `chi2`, `infovalue`, and `entropy`.

Internally, the split algorithm proceeds as follows:

1. All categories are merged into a single bin.
2. At the first iteration, all potential cutpoint indices are tested to see which one results in the maximum increase in the information function (`Gini`, `InfoValue`, `Entropy`, or `Chi2`). That cutpoint is then selected, and the bin is split.
3. The same procedure is reiterated for the next sub-bins.
4. The algorithm stops when the maximum number of bins is reached or when the splitting does not result in any additional change in the information change function.

The following table for a categorical predictor summarizes the values of the change function at each iteration. In this example, `'Gini'` is the measure of choice, such that the goal is to see a decrease of the Gini measure at each iteration.

Iteration 0 Bin Number	Member	Gini	Iteration 1 Bin Number	Member	Gini	Iteration 2 Bin Number	Member	Gini
1	'Tenant'		1	'Tenant'		1	'Tenant'	0.45638
1	'Subletter '		1	'Subletter '	0.44789	1	'Subletter '	
1	'Home Owner'		1	'Home Owner'		2	'Home Owner'	0.43984
1	'Other'		2	'Other'	0.41015	3	'Other'	0.41015

Iteration 0 Bin Number	Member	Gini	Iteration 1 Bin Number	Member	Gini	Iteration 2 Bin Number	Member	Gini
Total Gini		0.442765			0.442102			0.441822
Relative Change		0			0.001498			0.002128

The relative change at iteration *i* is with respect to the Gini measure of the entire bins at iteration *i*-1. The final result corresponds to that from the last iteration which, in this example, is iteration 2.

The following table for a numeric predictor summarizes the values of the change function at each iteration. In this example, `'Gini'` is the measure of choice, such that the goal is to see a decrease of the Gini measure at each iteration. Since most numeric predictors in datasets contain many bins, there is a preprocessing step where the data is pre-binned into 50 equal-frequency bins. This makes the pool of valid cutpoints to choose from for splitting smaller and more manageable.

Iteration 0 Bin Number	Member	Gini	Iteration 1 Bin Number	Gini	Iteration 2 Bin Number	Gini	Iteration 3 Bin Number	Gini
1	`'21'`		`'[-Inf,47]'`	0.473897	`'[-Inf,47]'`	0.473897	`'[-Inf,35]'`	0.494941
1	`'22'`		`'[47,Inf]'`	0.385238	`'[47,61]'`	0.407072	`'[35,47]'`	0.463201
1	`'23'`				`'[61,Inf]'`	0.208795	`'[47,61]'`	0.407072
1	`'74'`			0			`'[61,Inf]'`	0.208795
Total Gini		0.442765		0.435035		0.432048		0.430511
Relative Change		0		0.01746		0.006867		0.0356

The resulting split must be such that the information function (content) increases. As such, the best split is the one that results in the maximum information gain. The information functions supported are:

- Gini: Each split results in an increase in the Gini Ratio, defined as:

  ```
  G_r = 1- G_hat/G_p
  ```

  `G_p` is the Gini measure of the parent node, that is, of the given bins/categories prior to splitting. `G_hat` is the weighted Gini measure for the current split:

  ```
  G_hat = Sum((nj/N) * Gini(j), j=1..m)
  ```

  where

  `nj` is the total number of observations in the *j*th bin.

  `N` is the total number of observations in the dataset.

`m` is the number of splits for the given variable.

`Gini(j)` is the Gini measure for the *j*th bin.

The Gini measure for the split/node *j* is:

```
Gini(j) = 1 - (Gj^2+Bj^2) / (nj)^2
```

where `Gj`, `Bj` = Number of Goods and Bads for bin *j*.

- `InfoValue`: The information value for each split results in an increase in the total information. The split that is retained is the one which results in the maximum gain, within the acceptable gain tolerance. The Information Value (IV) for a given observation *j* is defined as:

  ```
  IV = sum( (pG_i-pB_i) * log(pG_i/pB_i), i=1..n)
  ```

  where

  `pG_i` is the distribution of Goods at observation `i`, that is `Goods(i)/Total_Goods`.

  `pB_i` is the distribution of Bads at observation `i`, that is `Bads(i)/Total_Bads`.

  `n` is the total number of bins.

- `Entropy`: Each split results in a decrease in entropy variance defined as:

  ```
   E = -sum(ni * Ei, i=1..n)
  ```

  where

  `ni` is the total count for bin `i`, that is `(ni = Gi + Bi)`.

  `Ei` is the entropy for row (or bin) `i`, defined as:

  ```
   Ei = -sum(Gi*log2(Gi/ni) + Bi*log2(Bi/ni))/N,
  i=1..n
  ```

- `Chi2`: Chi2 is computed pairwise for each pair of bins and measures the statistical difference between two groups. Splitting is selected at a point (cutpoint or category indexing) where the maximum Chi2 value is:

  ```
   Chi2 = sum(sum((Aij - Eij)^2/Eij , j=1..k), i=m,m+1)
  ```

  where

  `m` takes values from `1 ... n-1`, where `n` is the number of bins.

  `k` is the number of classes. Here `k = 2` for the (Goods, Bads).

  `Aij` is the number of observations in bin `i`, `j`th class.

  `Eij` is the expected frequency of `Aij`, which is equal to `(Ri*Cj)/N`.

  `Ri` is the number of observations in bin `i`, which is equal to `sum(Aij, j=1..k)`.

  `Cj` is the number of observations in the `j`th class, which is equal to `sum(Aij, I = m,m+1)`.

  `N` is the total number of observations, which is equal to `sum(Cj, j=1..k)`.

The Chi2 measure for the entire sample (as opposed to the pairwise Chi2 measure for adjacent bins) is:

```
Chi2 = sum(sum((Aij - Eij)^2/Eij , j=1..k), i=1..n)
```

Merge

Merge is a supervised automatic binning algorithm, where a measure is used to merge bins into buckets. The supported measures are chi2, gini, infovalue, and entropy.

Internally, the merge algorithm proceeds as follows:

1. All categories are initially in separate bins.
2. The user selected information function (Chi2, Gini, InfoValue or Entropy) is computed for any pair of adjacent bins.
3. At each iteration, the pair with the smallest information change measured by the selected information function is merged.
4. The merging continues until either:

   a. All pairwise information values are greater than the threshold set by the significance level or the relative change is smaller than the tolerance.

   b. If at the end, the number of bins is still greater than the MaxNumBins allowed, merging is forced until there are at most MaxNumBins bins. Similarly, merging stops when there are only MinNumBins bins.
5. For categorical, original bins/categories are pre-sorted according to the sorting of choice set by the user. For numeric data, the data is preprocessed to get IntialNumBins bins of equal frequency before the merging algorithm starts.

The following table for a categorical predictor summarizes the values of the change function at each iteration. In this example, 'Chi2' is the measure of choice. The default sorting by Odds is applied as a preprocessing step. The Chi2 value reported below at row *i* is for bins *i* and *i*+1. The significance level is 0.9 (90%), so that the inverse Chi2 value is 2.705543. This is the threshold below which adjacent pairs of bins are merged. The minimum number of bins is 2.

Iteration 0 Bin Number	Member	Chi2	Iteration 1 Bin Number	Member	Chi2	Iteration 2 Bin Number	Member	Chi2
1	'Tenant'	1.007613	1	'Tenant'	0.795920	1	'Tenant'	
2	'Subletter'	0.257347	2	'Subletter'		1	'Subletter'	
3	'Home Owner'	1.566330	2	'Home Owner'	1.522914	1	'Home Owner'	1.797395
4	'Other'		3	'Other'		2	'Other'	
Total Chi2		2.573943			2.317717			1.797395

The following table for a numeric predictor summarizes the values of the change function at each iteration. In this example, 'Chi2' is the measure of choice.

Iteration 0 Bin Number	Chi2	Iteration 1 Bins	Chi2		Final Iteration Bins	Chi2
'[-Inf,22]'	0.11814	'[-Inf,22]'	0.11814		'[-Inf,33]'	8.4876
'[22,23]'	1.6464	'[22,23]'	1.6464		'[33, 48]'	7.9369
...		...			'[48,64]'	9.956
'[58,59]'	0.311578	'[58,59]'	0.27489		'[64,65]'	9.6988
'*[59,60]*'	*0.068978*	'*[59,61]*'	1.8403		'[65,Inf]'	NaN
'[60,61]'	1.8709	'[61,62]'	5.7946	...		
'[61,62]'	5.7946	...				
...		'[69,70]'	6.4271			
'[69,70]'	6.4271	'[70,Inf]'	NaN			
'[70,Inf]'	NaN					
Total Chi2	67.467		67.399			23.198

The resulting merging must be such that any pair of adjacent bins is statistically different from each other, according to the chosen measure. The measures supported for `Merge` are:

- `Chi2`: Chi2 is computed pairwise for each pair of bins and measures the statistical difference between two groups. Merging is selected at a point (cutpoint or category indexing) where the maximum Chi2 value is:

  ```
  Chi2 = sum(sum((Aij - Eij)^2/Eij , j=1..k), i=m,m+1)
  ```

  where

  `m` takes values from `1 ... n-1`, and `n` is the number of bins.

  `k` is the number of classes. Here `k = 2` for the (Goods, Bads).

  `Aij` is the number of observations in bin `i`, `j`th class.

  `Eij` is the expected frequency of `Aij`, which is equal to `(Ri*Cj)/N`.

  `Ri` is the number of observations in bin `i`, which is equal to `sum(Aij, j=1..k)`.

  `Cj` is the number of observations in the `j`th class, which is equal to `sum(Aij, I = m,m+1)`.

  `N` is the total number of observations, which is equal to `sum(Cj, j=1..k)`.

  The `Chi2` measure for the entire sample (as opposed to the pairwise `Chi2` measure for adjacent bins) is:

  ```
  Chi2 = sum(sum((Aij - Eij)^2/Eij , j=1..k), i=1..n)
  ```
- Gini: Each merge results in a decrease in the Gini Ratio, defined as:

  `G_r = 1- G_hat/G_p`

`G_p` is the Gini measure of the parent node, that is, of the given bins/categories prior to merging. `G_hat` is the weighted Gini measure for the current merge:

```
G_hat = Sum((nj/N) * Gini(j), j=1..m)
```

where

`nj` is the total number of observations in the *j*th bin.

`N` is the total number of observations in the dataset.

`m` is the number of merges for the given variable.

`Gini(j)` is the Gini measure for the *j*th bin.

The Gini measure for the merge/node *j* is:

```
Gini(j) = 1 - (Gj^2+Bj^2) / (nj)^2
```

where `Gj`, `Bj` = Number of Goods and Bads for bin *j*.

- `InfoValue`: The information value for each merge will result in a decrease in the total information. The merge that is retained is the one which results in the minimum gain, within the acceptable gain tolerance. The Information Value (IV) for a given observation *j* is defined as:

  ```
  IV = sum( (pG_i-pB_i) * log(pG_i/pB_i), i=1..n)
  ```

  where

  `pG_i` is the distribution of Goods at observation `i`, that is `Goods(i)/Total_Goods`.

  `pB_i` is the distribution of Bads at observation `i`, that is `Bads(i)/Total_Bads`.

  `n` is the total number of bins.
- `Entropy`: Each merge results in an increase in entropy variance defined as:

  ```
   E = -sum(ni * Ei, i=1..n)
  ```

  where

  `ni` is the total count for bin `i`, that is `(ni = Gi + Bi)`.

  `Ei` is the entropy for row (or bin) `i`, defined as:

  ```
   Ei = -sum(Gi*log2(Gi/ni) + Bi*log2(Bi/ni))/N,
  i=1..n
  ```

Note When using the Merge algorithm, if there are pure bins (bins that have either zero count of `Goods` or zero count of `Bads`), the statistics such as Information Value and Entropy have non-finite values. To account for this, a frequency shift of `.5` is applied for computing various statistics whenever the algorithm finds pure bins.

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 15-1836 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 15-1836 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 8-54.

Version History

Introduced in R2014b

References

[1] Anderson, R. *The Credit Scoring Toolkit*. Oxford University Press, 2007.

[2] Kerber, R. "ChiMerge: Discretization of Numeric Attributes." *AAAI-92 Proceedings.* 1992.

[3] Liu, H., et. al. *Data Mining, Knowledge, and Discovery.* Vol 6. Issue 4. October 2002, pp. 393-423.

[4] Refaat, M. *Data Preparation for Data Mining Using SAS.* Morgan Kaufmann, 2006.

[5] Refaat, M. *Credit Risk Scorecards: Development and Implementation Using SAS.* lulu.com, 2011.

[6] Thomas, L., et al. *Credit Scoring and Its Applications.* Society for Industrial and Applied Mathematics, 2002.

See Also

`creditscorecard` | `bininfo` | `predictorinfo` | `modifypredictor` | `modifybins` | `bindata` | `plotbins` | `fitmodel` | `displaypoints` | `formatpoints` | `score` | `setmodel` | `probdefault` | `validatemodel`

Topics

“Case Study for Credit Scorecard Analysis” on page 8-70
“Credit Scorecard Modeling with Missing Values” on page 8-56
“Troubleshooting Credit Scorecard Results” on page 8-63
“Credit Scorecard Modeling Workflow” on page 8-51
“About Credit Scorecards” on page 8-47
“Credit Scorecard Modeling Using Observation Weights” on page 8-54

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: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIn
        NumericPredictors: {'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBalance'  'Util
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
           BinMissingData: 0
                    IDVar: 'CustID'
            PredictorVars: {'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIncome'  'Tm
                     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-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 + AMBala
    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
```

Compute Probability for Default Using Credit ScoreCard Data When Using the 'BinMissingData' Option

This example describes both the assignment of points for missing data when the 'BinMissingData' option is set to true, and the corresponding computation of probabilities of default.

- Predictors that have missing data in the training set have an explicit bin for <missing> with corresponding points in the final scorecard. These points are computed from the Weight-of-Evidence (WOE) value for the <missing> bin and the logistic model coefficients. For scoring purposes, these points are assigned to missing values and to out-of-range values, and the final score is mapped to a probability of default when using probdefault.
- Predictors with no missing data in the training set have no <missing> bin, therefore no WOE can be estimated from the training data. By default, the points for missing and out-of-range values are set to NaN, and this leads to a score of NaN when running score. For predictors that have no explicit <missing> bin, use the name-value argument 'Missing' in formatpoints to indicate how missing data should be treated for scoring purposes. The final score is then mapped to a probability of default when using probdefault.

Create a creditscorecard object using the CreditCardData.mat file to load the dataMissing with missing values.

```
load CreditCardData.mat
head(dataMissing,5)
```

```
    CustID    CustAge    TmAtAddress     ResStatus     EmpStatus    CustIncome    TmWBank    Othe
    ______    _______    ___________    ___________    _________    __________    _______    ____

      1          53          62         <undefined>    Unknown        50000         55         Ye
      2          61          22         Home Owner     Employed       52000         25         Ye
      3          47          30         Tenant         Employed       37000         61         No
      4         NaN          75         Home Owner     Employed       53000         20         Ye
      5          68          56         Home Owner     Employed       53000         14         Ye
```

Use creditscorecard with the name-value argument 'BinMissingData' set to true to bin the missing numeric or categorical data in a separate bin. Apply automatic binning.

```
sc = creditscorecard(dataMissing,'IDVar','CustID','BinMissingData',true);
sc = autobinning(sc);

disp(sc)
```

```
  creditscorecard with properties:

                GoodLabel: 0
              ResponseVar: 'status'
               WeightsVar: ''
                 VarNames: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIn
        NumericPredictors: {'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBalance'  'Uti
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
           BinMissingData: 1
                    IDVar: 'CustID'
            PredictorVars: {'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIncome'  'Tr
                     Data: [1200x11 table]
```

Set a minimum value of 0 for CustAge and CustIncome. With this, any negative age or income information becomes invalid or "out-of-range". For scoring and probability of default computations, out-of-range values are given the same points as missing values.

```
sc = modifybins(sc,'CustAge','MinValue',0);
sc = modifybins(sc,'CustIncome','MinValue',0);
```

Display bin information for numeric data for `'CustAge'` that includes missing data in a separate bin labelled `<missing>`.

```
bi = bininfo(sc,'CustAge');
disp(bi)
```

```
         Bin           Good    Bad     Odds        WOE       InfoValue 
    _____________      ____    ___    ______    ________    __________

    {'[0,33)'   }       69     52     1.3269    -0.42156      0.018993
    {'[33,37)'  }       63     45        1.4    -0.36795      0.012839
    {'[37,40)'  }       72     47     1.5319     -0.2779     0.0079824
    {'[40,46)'  }      172     89     1.9326    -0.04556     0.0004549
    {'[46,48)'  }       59     25       2.36     0.15424     0.0016199
    {'[48,51)'  }       99     41     2.4146     0.17713     0.0035449
    {'[51,58)'  }      157     62     2.5323     0.22469     0.0088407
    {'[58,Inf]' }       93     25       3.72     0.60931      0.032198
    {'<missing>'}       19     11     1.7273    -0.15787    0.00063885
    {'Totals'   }      803    397     2.0227         NaN      0.087112
```

Display bin information for categorical data for `'ResStatus'` that includes missing data in a separate bin labelled `<missing>`.

```
bi = bininfo(sc,'ResStatus');
disp(bi)
```

```
         Bin             Good    Bad     Odds         WOE        InfoValue 
    ______________       ____    ___    ______    _________    __________

    {'Tenant'     }      296     161    1.8385    -0.095463     0.0035249
    {'Home Owner'}       352     171    2.0585     0.017549    0.00013382
    {'Other'      }      128      52    2.4615      0.19637     0.0055808
    {'<missing>' }        27      13    2.0769     0.026469    2.3248e-05
    {'Totals'     }      803     397    2.0227          NaN     0.0092627
```

For the `'CustAge'` and `'ResStatus'` predictors, there is missing data (`NaN`s and `<undefined>`) in the training data, and the binning process estimates a WOE value of -0.15787 and 0.026469 respectively for missing data in these predictors, as shown above.

For `EmpStatus` and `CustIncome` there is no explicit bin for missing values because the training data has no missing values for these predictors.

```
bi = bininfo(sc,'EmpStatus');
disp(bi)
```

```
        Bin           Good    Bad     Odds        WOE       InfoValue
    ____________      ____    ___    ______    ________    _________

    {'Unknown' }      396     239    1.6569    -0.19947    0.021715 
    {'Employed'}      407     158    2.5759      0.2418    0.026323 
    {'Totals'  }      803     397    2.0227         NaN    0.048038 
```

```
bi = bininfo(sc,'CustIncome');
disp(bi)
```

```
           Bin             Good    Bad     Odds         WOE        InfoValue 
    _________________      ____    ___    _______    _________    __________
    {'[0,29000)'    }     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 `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). For predictors that have missing data, there is an explicit `<missing>` bin, with a corresponding WOE value computed from the data. When using `fitmodel`, the corresponding WOE value for the <missing> bin is applied when performing the WOE transformation.

```
[sc,mdl] = 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 = 1442.8477, Chi2Stat = 4.4974731, PValue = 0.033944979
6. Adding ResStatus, Deviance = 1438.9783, Chi2Stat = 3.86941, PValue = 0.049173805
7. Adding OtherCC, Deviance = 1434.9751, Chi2Stat = 4.0031966, PValue = 0.045414057

Generalized linear regression model:
    logit(status) ~ 1 + CustAge + ResStatus + EmpStatus + CustIncome + TmWBank + OtherCC + AMBala
    Distribution = Binomial

Estimated Coefficients:
                   Estimate       SE       tStat       pValue  
                   ________    ________    ______    __________

    (Intercept)    0.70229     0.063959     10.98    4.7498e-28
    CustAge        0.57421      0.25708    2.2335      0.025513
    ResStatus       1.3629      0.66952    2.0356       0.04179
    EmpStatus      0.88373       0.2929    3.0172      0.002551
    CustIncome     0.73535       0.2159     3.406    0.00065929
    TmWBank         1.1065      0.23267    4.7556    1.9783e-06
    OtherCC         1.0648      0.52826    2.0156      0.043841
    AMBalance       1.0446      0.32197    3.2443     0.0011775


1200 observations, 1192 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 88.5, p-value = 2.55e-16
```

Scale the scorecard points by the "points, odds, and points to double the odds (PDO)" method using the `'PointsOddsAndPDO'` argument of `formatpoints`. 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).

Display the scorecard showing the scaled points for predictors retained in the fitting model.

```
sc = formatpoints(sc,'PointsOddsAndPDO',[500 2 50]);
PointsInfo = displaypoints(sc)
```

```
PointsInfo=38×3 table
     Predictors           Bin           Points
    _____________    ______________    ______

    {'CustAge'  }    {'[0,33)'    }    54.062
    {'CustAge'  }    {'[33,37)'   }    56.282
    {'CustAge'  }    {'[37,40)'   }    60.012
    {'CustAge'  }    {'[40,46)'   }    69.636
    {'CustAge'  }    {'[46,48)'   }    77.912
    {'CustAge'  }    {'[48,51)'   }     78.86
    {'CustAge'  }    {'[51,58)'   }     80.83
    {'CustAge'  }    {'[58,Inf]'  }     96.76
    {'CustAge'  }    {'<missing>' }    64.984
    {'ResStatus'}    {'Tenant'    }    62.138
    {'ResStatus'}    {'Home Owner'}    73.248
    {'ResStatus'}    {'Other'     }    90.828
    {'ResStatus'}    {'<missing>' }    74.125
    {'EmpStatus'}    {'Unknown'   }    58.807
    {'EmpStatus'}    {'Employed'  }    86.937
    {'EmpStatus'}    {'<missing>' }       NaN
          ⋮
```

Notice that points for the `<missing>` bin for `CustAge` and `ResStatus` are explicitly shown (as `64.9836` and `74.1250`, respectively). These points are computed from the WOE value for the <missing> bin and the logistic model coefficients.

For predictors that have no missing data in the training set, there is no explicit <missing> bin. By default the points are set to `NaN` for missing data, and they lead to a score of `NaN` when running `score`. For predictors that have no explicit <missing> bin, use the name-value argument `'Missing'` in `formatpoints` to indicate how missing data should be treated for scoring purposes.

For the purpose of illustration, take a few rows from the original data as test data and introduce some missing data. Also introduce some invalid, or out-of-range, values. For numeric data, values below the minimum (or above the maximum) allowed are considered invalid, such as a negative value for age (recall `'MinValue'` was earlier set to 0 for `CustAge` and `CustIncome`). For categorical data, invalid values are categories not explicitly included in the scorecard, for example, a residential status not previously mapped to scorecard categories, such as "House", or a meaningless string such as "abc123".

```
tdata = dataMissing(11:18,mdl.PredictorNames); % Keep only the predictors retained in the model
% Set some missing values
tdata.CustAge(1) = NaN;
tdata.ResStatus(2) = '<undefined>';
tdata.EmpStatus(3) = '<undefined>';
tdata.CustIncome(4) = NaN;
% Set some invalid values
tdata.CustAge(5) = -100;
tdata.ResStatus(6) = 'House';
tdata.EmpStatus(7) = 'Freelancer';
tdata.CustIncome(8) = -1;
disp(tdata)
```

```
    CustAge     ResStatus       EmpStatus      CustIncome    TmWBank    OtherCC    AMBalance
    _______    ___________    ___________    __________    _______    _______    _________

      NaN      Tenant         Unknown          34000         44         Yes        119.8
```

```
     48      <undefined>    Unknown          44000         14          Yes        403.62
     65      Home Owner     <undefined>      48000          6          No         111.88
     44      Other          Unknown            NaN         35          No         436.41
   -100      Other          Employed         46000         16          Yes        162.21
     33      House          Employed         36000         36          Yes        845.02
     39      Tenant         Freelancer       34000         40          Yes        756.26
     24      Home Owner     Employed            -1         19          Yes        449.61
```

Score the new data and see how points are assigned for missing `CustAge` and `ResStatus`, because we have an explicit bin with points for `<missing>`. However, for `EmpStatus` and `CustIncome` the `score` function sets the points to `NaN`. The corresponding probabilities of default are also set to `NaN`.

```
[Scores,Points] = score(sc,tdata);
disp(Scores)
```

```
  481.2231
  520.8353
       NaN
       NaN
  551.7922
  487.9588
       NaN
       NaN
```

```
disp(Points)
```

```
    CustAge    ResStatus    EmpStatus    CustIncome    TmWBank    OtherCC    AMBalance
    _______    _________    _________    __________    _______    _______    _________

    64.984      62.138       58.807        67.893      61.858     75.622      89.922
     78.86      74.125       58.807        82.439      61.061     75.622      89.922
     96.76      73.248          NaN        96.969      51.132     50.914      89.922
    69.636      90.828       58.807           NaN      61.858     50.914      89.922
    64.984      90.828       86.937        82.439      61.061     75.622      89.922
    56.282      74.125       86.937        70.107      61.858     75.622      63.028
    60.012      62.138          NaN        67.893      61.858     75.622      63.028
    54.062      73.248       86.937           NaN      61.061     75.622      89.922
```

```
pd = probdefault(sc,tdata);
disp(pd)
```

```
    0.3934
    0.2725
       NaN
       NaN
    0.1961
    0.3714
       NaN
       NaN
```

Use the name-value argument `'Missing'` in `formatpoints` to choose how to assign points to missing values for predictors that do not have an explicit `<missing>` bin. In this example, use the `'MinPoints'` option for the `'Missing'` argument. The minimum points for `EmpStatus` in the scorecard displayed above are `58.8072`, and for `CustIncome` the minimum points are `29.3753`. All rows now have a score and a corresponding probability of default.

```
sc = formatpoints(sc,'Missing','MinPoints');
[Scores,Points] = score(sc,tdata);
disp(Scores)
```

```
  481.2231
  520.8353
  517.7532
  451.3405
  551.7922
  487.9588
  449.3577
  470.2267
```

```
disp(Points)
```

```
    CustAge    ResStatus    EmpStatus    CustIncome    TmWBank    OtherCC    AMBalance
    _______    _________    _________    __________    _______    _______    _________

    64.984      62.138       58.807        67.893      61.858     75.622      89.922  
     78.86      74.125       58.807        82.439      61.061     75.622      89.922  
     96.76      73.248       58.807        96.969      51.132     50.914      89.922  
    69.636      90.828       58.807        29.375      61.858     50.914      89.922  
    64.984      90.828       86.937        82.439      61.061     75.622      89.922  
    56.282      74.125       86.937        70.107      61.858     75.622      63.028  
    60.012      62.138       58.807        67.893      61.858     75.622      63.028  
    54.062      73.248       86.937        29.375      61.061     75.622      89.922  
```

```
pd = probdefault(sc,tdata);
disp(pd)
```

```
    0.3934
    0.2725
    0.2810
    0.4954
    0.1961
    0.3714
    0.5022
    0.4304
```

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.

More About

Default Probability

After the unscaled scores are computed (see “Algorithms for Computing and Scaling Scores” on page 15-1662), 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
```

Version History

Introduced in R2015a

References

[1] Refaat, M. *Credit Risk Scorecards: Development and Implementation Using SAS.* lulu.com, 2011.

See Also

`creditscorecard` | `bininfo` | `predictorinfo` | `modifypredictor` | `modifybins` | `bindata` | `plotbins` | `fitmodel` | `displaypoints` | `formatpoints` | `score` | `setmodel` | `validatemodel` | `table`

Topics

“Case Study for Credit Scorecard Analysis” on page 8-70
“Troubleshooting Credit Scorecard Results” on page 8-63
“Credit Scorecard Modeling Workflow” on page 8-51
“About Credit Scorecards” on page 8-47

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: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustI
        NumericPredictors: {'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBalance'  'Uti
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
           BinMissingData: 0
                    IDVar: 'CustID'
            PredictorVars: {'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIncome'  'Tr
                     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-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 + AMBala
    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.

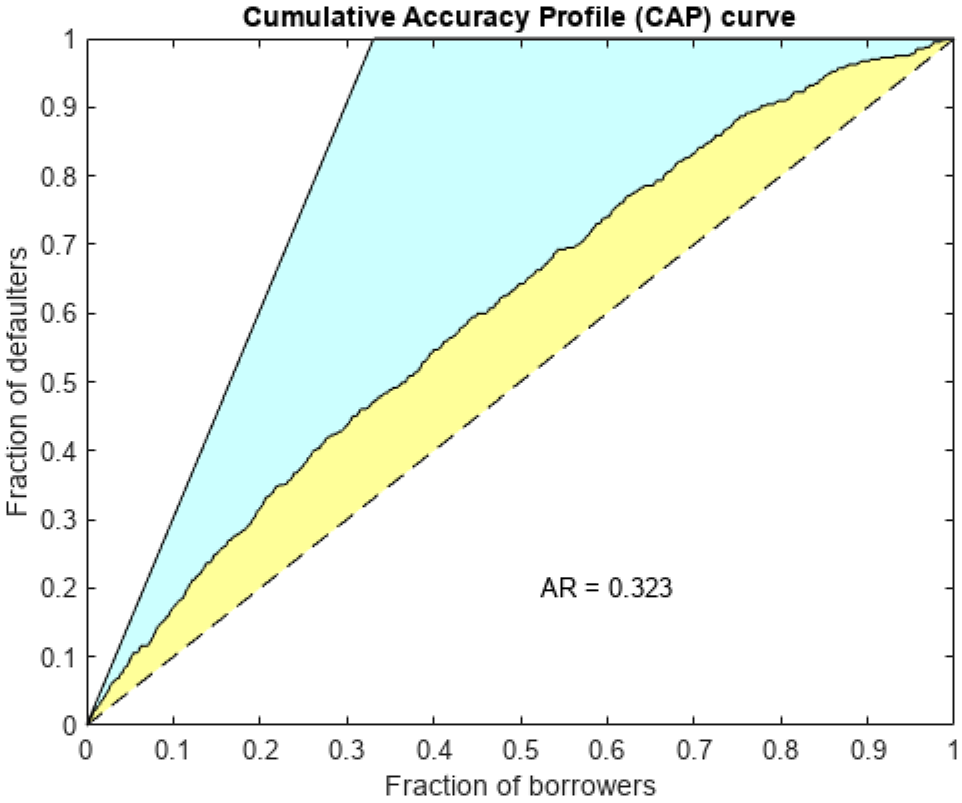
```
sc = formatpoints(sc, 'PointsOddsAndPDO',[500,2,50]);
```

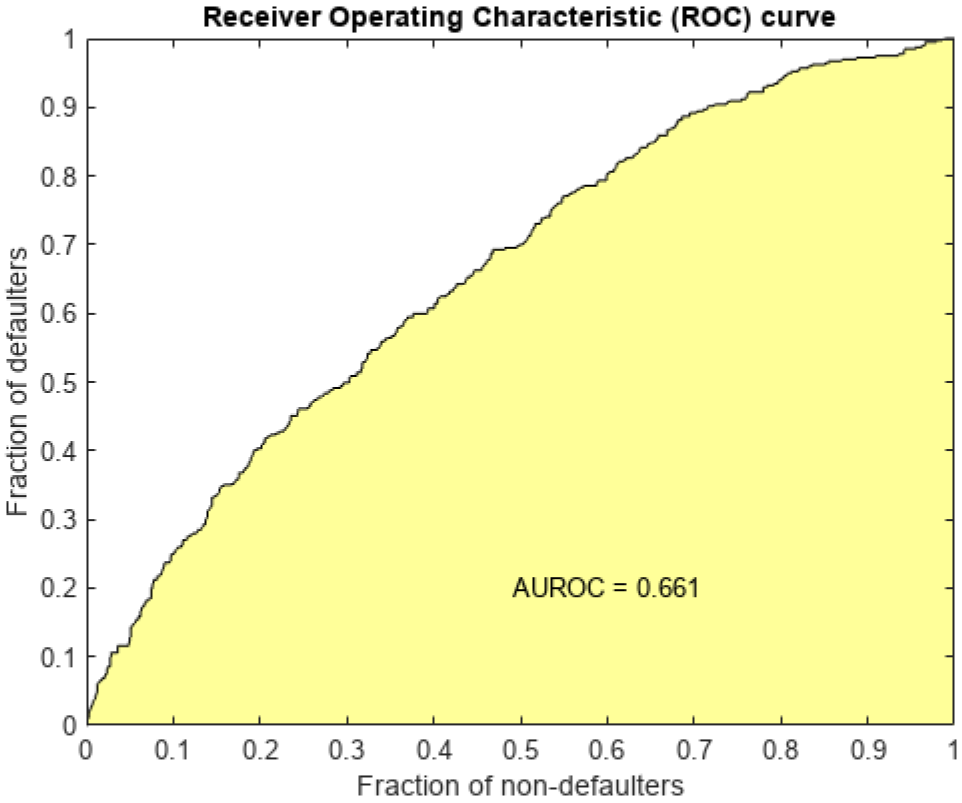
Score the data.

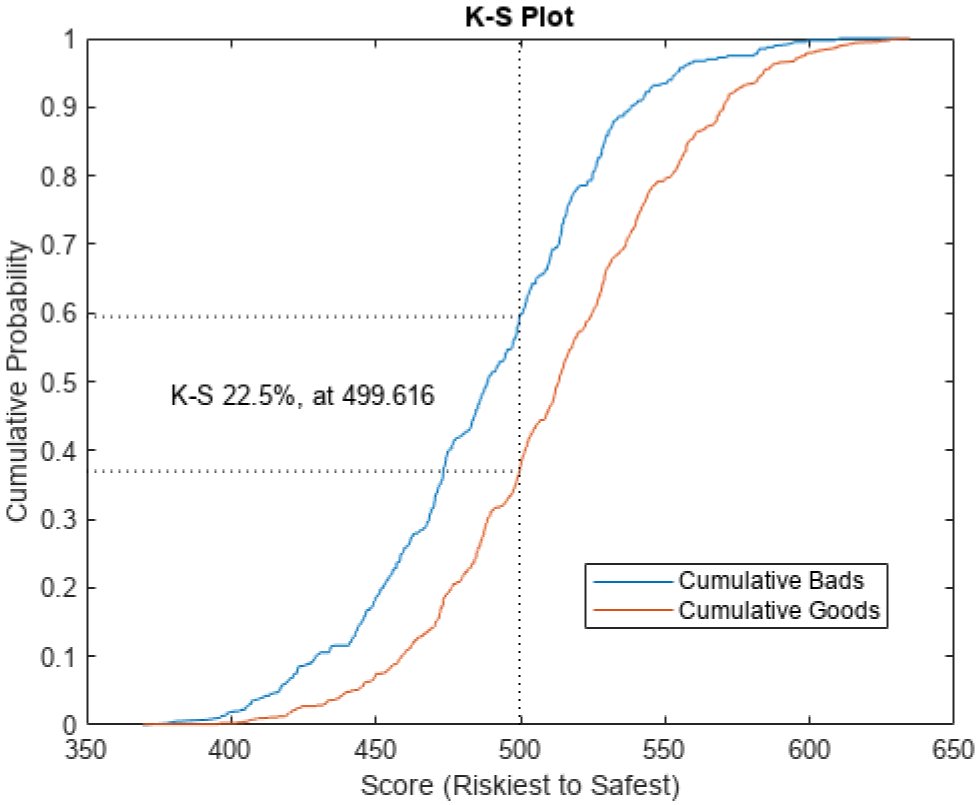
```
scores = score(sc);
```

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    Sensitivity
    ______     ___________    ________    _________    _________    __________    ___________

    369.54       0.75313          0           1           802          397                 0
    378.19       0.73016          1           1           802          396         0.0025189
    380.28       0.72444          2           1           802          395         0.0050378
    391.49       0.69234          3           1           802          394         0.0075567
    395.57       0.68017          4           1           802          393          0.010076
    396.14       0.67846          4           2           801          393          0.010076
    396.45       0.67752          5           2           801          392          0.012594
    398.61       0.67094          6           2           801          391          0.015113
    398.68       0.67072          7           2           801          390          0.017632
    401.33       0.66255          8           2           801          389          0.020151
    402.66       0.65842          8           3           800          389          0.020151
    404.25       0.65346          9           3           800          388           0.02267
    404.73       0.65193          9           4           799          388           0.02267
    405.53      0.64941         11             4           799           386          0.027708
     405.7      0.64887         11             5           798           386          0.027708
```

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: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustI
        NumericPredictors: {'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBalance'  'Uti
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
           BinMissingData: 0
                    IDVar: 'CustID'
            PredictorVars: {'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIncome'  'T
                     Data: [1200x12 table]
```

Perform automatic binning.

```
sc = autobinning(sc)
```

```
sc = 
  creditscorecard with properties:

                GoodLabel: 0
              ResponseVar: 'status'
               WeightsVar: 'RowWeights'
                 VarNames: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustI
        NumericPredictors: {'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBalance'  'Uti
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
           BinMissingData: 0
                    IDVar: 'CustID'
            PredictorVars: {'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIncome'  'T
                     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.

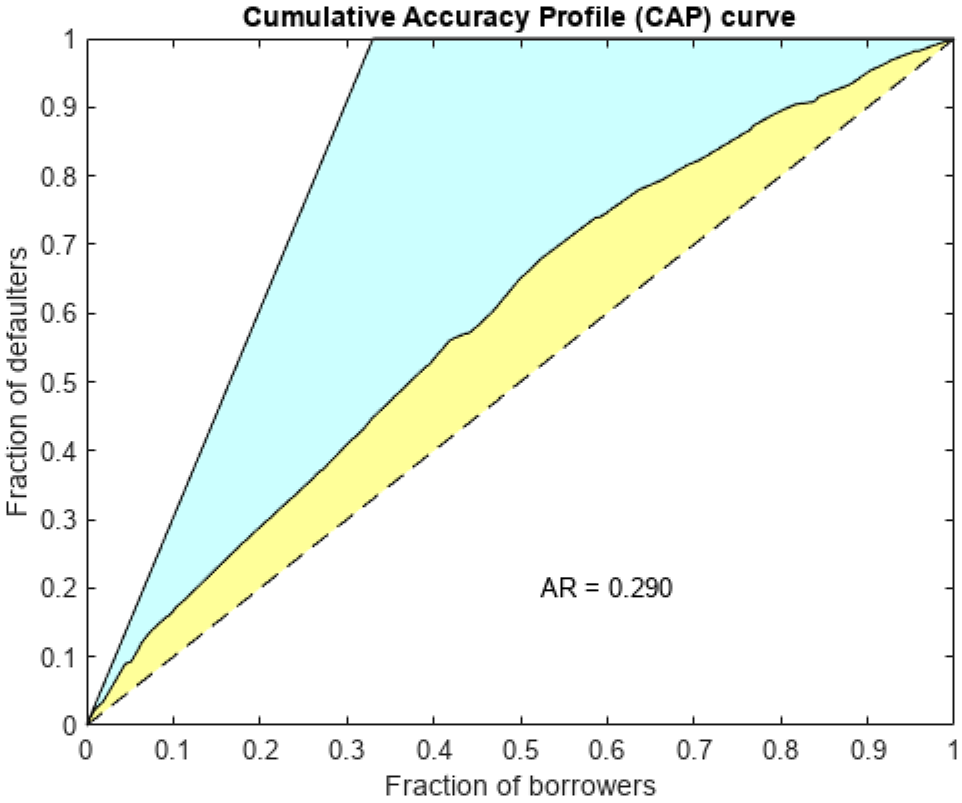
```
sc = formatpoints(sc, 'PointsOddsAndPDO',[500,2,50]);
```

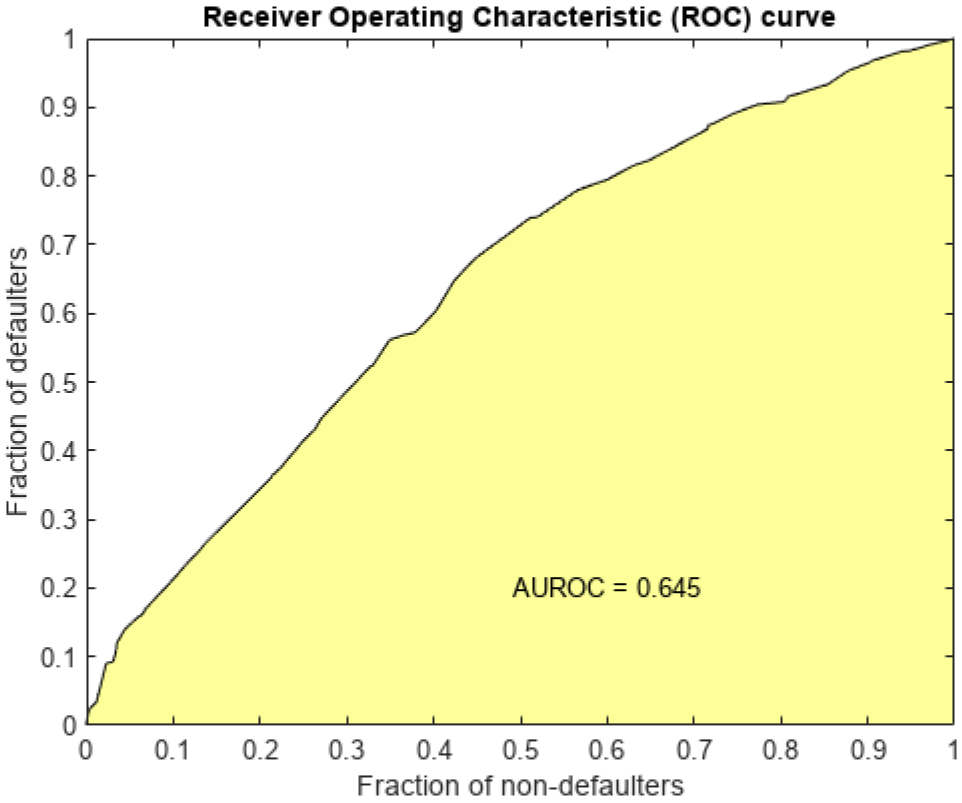
Score the data.

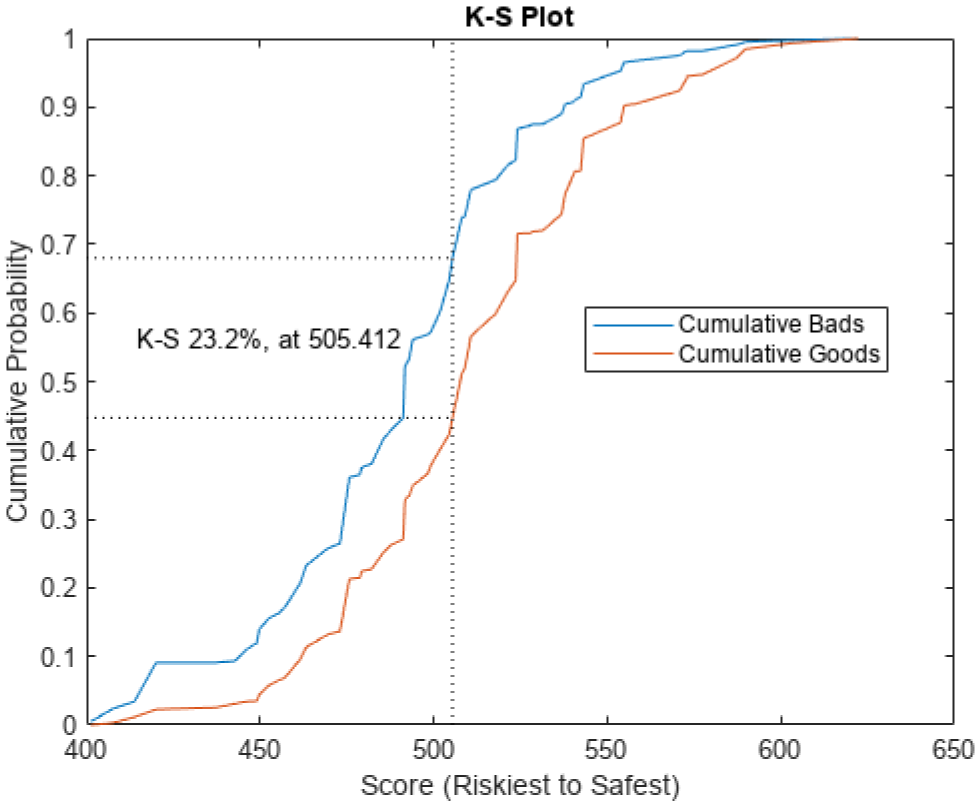
```
scores = score(sc);
```

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'});
```


Cumulative Accuracy Profile (CAP) curve
Fraction of defaulters
Fraction of borrowers
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
AR = 0.290


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



```
Stats
```

```
Stats=4×2 table
              Measure                Value
    _________________________       _______

    {'Accuracy Ratio'       }       0.28972
    {'Area under ROC curve'}       0.64486
    {'KS statistic'         }       0.23215
    {'KS score'             }        505.41
```

```
T(1:10,:)
```

```
ans=10×9 table
    Scores     ProbDefault     TrueBads     FalseBads     TrueGoods     FalseGoods     Sensitivity
    ______     ___________     ________     _________     _________     __________     ___________

    401.34       0.66253        1.0788            0        411.95         201.95        0.0053135
    407.59       0.64289        4.8363       1.2768        410.67         198.19         0.023821
    413.79       0.62292        6.9469       4.6942        407.25         196.08         0.034216
    420.04       0.60236        18.459       9.3899        402.56         184.57         0.090918
    437.27         0.544        18.459       10.514        401.43         184.57         0.090918
    442.83       0.52481        18.973       12.794        399.15         184.06         0.093448
    446.19       0.51319        22.396        14.15         397.8         180.64          0.11031
    449.08       0.50317        24.325       14.405        397.54         178.71          0.11981
    449.73       0.50095        28.246       18.049         393.9         174.78          0.13912
    452.44      0.49153       31.511      23.565       388.38        171.52          0.1552
```

Validate a Credit Score Card Model When Using the 'BinMissingData' Option

This example describes both the assignment of points for missing data when the `'BinMissingData'` option is set to `true`, and the corresponding computation of model validation statistics.

- Predictors that have missing data in the training set have an explicit bin for `<missing>` with corresponding points in the final scorecard. These points are computed from the Weight-of-Evidence (WOE) value for the `<missing>` bin and the logistic model coefficients. For scoring purposes, these points are assigned to missing values and to out-of-range values, and the final score is used to compute model validation statistics with `validatemodel`.
- Predictors with no missing data in the training set have no `<missing>` bin, therefore no WOE can be estimated from the training data. By default, the points for missing and out-of-range values are set to `NaN`, and this leads to a score of `NaN` when running `score`. For predictors that have no explicit `<missing>` bin, use the name-value argument `'Missing'` in `formatpoints` to indicate how missing data should be treated for scoring purposes. The final score is used to compute model validation statistics with `validatemodel`.

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `dataMissing` with missing values.

```
load CreditCardData.mat
head(dataMissing,5)
```

```
    CustID    CustAge    TmAtAddress     ResStatus     EmpStatus    CustIncome    TmWBank    Othe
    ______    _______    ___________    ___________    _________    __________    _______    ____

      1          53           62        <undefined>    Unknown        50000          55        Ye
      2          61           22        Home Owner     Employed       52000          25        Ye
      3          47           30        Tenant         Employed       37000          61        No
      4         NaN           75        Home Owner     Employed       53000          20        Ye
      5          68           56        Home Owner     Employed       53000          14        Ye
```

Use `creditscorecard` with the name-value argument `'BinMissingData'` set to `true` to bin the missing numeric or categorical data in a separate bin. Apply automatic binning.

```
sc = creditscorecard(dataMissing,'IDVar','CustID','BinMissingData',true);
sc = autobinning(sc);

disp(sc)
```

```
  creditscorecard with properties:

                GoodLabel: 0
              ResponseVar: 'status'
               WeightsVar: ''
                 VarNames: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustI
        NumericPredictors: {'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBalance'  'Uti
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
           BinMissingData: 1
                    IDVar: 'CustID'
            PredictorVars: {'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIncome'  'Tr
                     Data: [1200x11 table]
```

Set a minimum value of zero for `CustAge` and `CustIncome`. With this, any negative age or income information becomes invalid or "out-of-range". For scoring and probability of default computations, out-of-range values are given the same points as missing values.

```
sc = modifybins(sc,'CustAge','MinValue',0);
sc = modifybins(sc,'CustIncome','MinValue',0);
```

Display bin information for numeric data for `'CustAge'` that includes missing data in a separate bin labelled `<missing>`.

```
bi = bininfo(sc,'CustAge');
disp(bi)
```

```
         Bin          Good    Bad     Odds        WOE       InfoValue 
    ______________    ____    ___    ______    ________    __________

    {'[0,33)'   }      69      52    1.3269    -0.42156      0.018993
    {'[33,37)'  }      63      45       1.4    -0.36795      0.012839
    {'[37,40)'  }      72      47    1.5319     -0.2779     0.0079824
    {'[40,46)'  }     172      89    1.9326    -0.04556     0.0004549
    {'[46,48)'  }      59      25      2.36     0.15424     0.0016199
    {'[48,51)'  }      99      41    2.4146     0.17713     0.0035449
    {'[51,58)'  }     157      62    2.5323     0.22469     0.0088407
    {'[58,Inf]' }      93      25      3.72     0.60931      0.032198
    {'<missing>'}      19      11    1.7273    -0.15787    0.00063885
    {'Totals'   }     803     397    2.0227         NaN      0.087112
```

Display bin information for categorical data for `'ResStatus'` that includes missing data in a separate bin labelled `<missing>`.

```
bi = bininfo(sc,'ResStatus');
disp(bi)
```

```
         Bin            Good    Bad     Odds        WOE         InfoValue 
    _______________    ____    ___    ______    _________    __________

    {'Tenant'     }     296     161    1.8385    -0.095463     0.0035249
    {'Home Owner'}     352     171    2.0585     0.017549    0.00013382
    {'Other'      }     128      52    2.4615      0.19637     0.0055808
    {'<missing>'  }      27      13    2.0769     0.026469    2.3248e-05
    {'Totals'     }     803     397    2.0227          NaN     0.0092627
```

For the `'CustAge'` and `'ResStatus'` predictors, there is missing data (`NaN`s and `<undefined>`) in the training data, and the binning process estimates a WOE value of `-0.15787` and `0.026469` respectively for missing data in these predictors, as shown above.

For `EmpStatus` and `CustIncome` there is no explicit bin for missing values, because the training data has no missing values for these predictors.

```
bi = bininfo(sc,'EmpStatus');
disp(bi)
```

```
        Bin           Good    Bad     Odds       WOE       InfoValue 
    ____________     ____    ___    ______    ________    _________
    {'Unknown' }    396     239    1.6569    -0.19947    0.021715
    {'Employed'}    407     158    2.5759      0.2418    0.026323
    {'Totals'  }    803     397    2.0227         NaN    0.048038
```

```
bi = bininfo(sc,'CustIncome');
disp(bi)
```

```
            Bin            Good    Bad     Odds        WOE       InfoValue 
    _________________    ____    ___    _______    _________    __________

    {'[0,29000)'    }     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 `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). For predictors that have missing data, there is an explicit `<missing>` bin, with a corresponding WOE value computed from the data. When using `fitmodel`, the corresponding WOE value for the <missing> bin is applied when performing the WOE transformation.

```
[sc,mdl] = 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 = 1442.8477, Chi2Stat = 4.4974731, PValue = 0.033944979
6. Adding ResStatus, Deviance = 1438.9783, Chi2Stat = 3.86941, PValue = 0.049173805
7. Adding OtherCC, Deviance = 1434.9751, Chi2Stat = 4.0031966, PValue = 0.045414057

Generalized linear regression model:
    logit(status) ~ 1 + CustAge + ResStatus + EmpStatus + CustIncome + TmWBank + OtherCC + AMBala
    Distribution = Binomial

Estimated Coefficients:
                   Estimate       SE        tStat       pValue  
                   ________    ________    ______    __________

    (Intercept)    0.70229     0.063959     10.98    4.7498e-28
    CustAge        0.57421      0.25708    2.2335      0.025513
    ResStatus       1.3629      0.66952    2.0356       0.04179
    EmpStatus      0.88373       0.2929    3.0172      0.002551
    CustIncome     0.73535       0.2159     3.406    0.00065929
    TmWBank         1.1065      0.23267    4.7556    1.9783e-06
    OtherCC         1.0648      0.52826    2.0156      0.043841
    AMBalance       1.0446      0.32197    3.2443     0.0011775


1200 observations, 1192 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 88.5, p-value = 2.55e-16
```

Scale the scorecard points by the "points, odds, and points to double the odds (PDO)" method using the `'PointsOddsAndPDO'` argument of `formatpoints`. 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).

Display the scorecard showing the scaled points for predictors retained in the fitting model.

```
sc = formatpoints(sc,'PointsOddsAndPDO',[500 2 50]);
PointsInfo = displaypoints(sc)
```

```
PointsInfo=38×3 table
     Predictors          Bin          Points
    _____________    ______________   ______

    {'CustAge'  }    {'[0,33)'    }   54.062
    {'CustAge'  }    {'[33,37)'   }   56.282
    {'CustAge'  }    {'[37,40)'   }   60.012
    {'CustAge'  }    {'[40,46)'   }   69.636
    {'CustAge'  }    {'[46,48)'   }   77.912
    {'CustAge'  }    {'[48,51)'   }    78.86
    {'CustAge'  }    {'[51,58)'   }    80.83
    {'CustAge'  }    {'[58,Inf]'  }    96.76
    {'CustAge'  }    {'<missing>' }   64.984
    {'ResStatus'}    {'Tenant'    }   62.138
    {'ResStatus'}    {'Home Owner'}   73.248
    {'ResStatus'}    {'Other'     }   90.828
    {'ResStatus'}    {'<missing>' }   74.125
    {'EmpStatus'}    {'Unknown'   }   58.807
    {'EmpStatus'}    {'Employed'  }   86.937
    {'EmpStatus'}    {'<missing>' }      NaN
      ⋮
```

Notice that points for the `<missing>` bin for `CustAge` and `ResStatus` are explicitly shown (as `64.9836` and `74.1250`, respectively). These points are computed from the WOE value for the `<missing>` bin, and the logistic model coefficients.

For predictors that have no missing data in the training set, there is no explicit `<missing>` bin. By default the points are set to `NaN` for missing data, and they lead to a score of `NaN` when running `score`. For predictors that have no explicit `<missing>` bin, use the name-value argument `'Missing'` in `formatpoints` to indicate how missing data should be treated for scoring purposes.

For the purpose of illustration, take a few rows from the original data as test data and introduce some missing data. Also introduce some invalid, or out-of-range, values. For numeric data, values below the minimum (or above the maximum) allowed are considered invalid, such as a negative value for age (recall `'MinValue'` was earlier set to 0 for `CustAge` and `CustIncome`). For categorical data, invalid values are categories not explicitly included in the scorecard, for example, a residential status not previously mapped to scorecard categories, such as "House", or a meaningless string such as "abc123".

This is a very small validation data set, only used to illustrate the scoring of rows with missing and out-of-range values, and its relationship with model validation.

```
tdata = dataMissing(11:18,mdl.PredictorNames); % Keep only the predictors retained in the model
tdata.status = dataMissing.status(11:18); % Copy the response variable value, needed for validat
% Set some missing values
tdata.CustAge(1) = NaN;
tdata.ResStatus(2) = '<undefined>';
tdata.EmpStatus(3) = '<undefined>';
tdata.CustIncome(4) = NaN;
% Set some invalid values
tdata.CustAge(5) = -100;
tdata.ResStatus(6) = 'House';
tdata.EmpStatus(7) = 'Freelancer';
tdata.CustIncome(8) = -1;
disp(tdata)
```

```
    CustAge     ResStatus      EmpStatus      CustIncome    TmWBank    OtherCC    AMBalance    sta
    _______    ___________    ___________    __________    _______    _______    _________    ___

      NaN      Tenant         Unknown          34000         44         Yes        119.8       1
       48      <undefined>    Unknown          44000         14         Yes       403.62       0
       65      Home Owner     <undefined>      48000          6         No        111.88       0
       44      Other          Unknown            NaN         35         No        436.41       0
     -100      Other          Employed         46000         16         Yes       162.21       0
       33      House          Employed         36000         36         Yes       845.02       0
       39      Tenant         Freelancer       34000         40         Yes       756.26       1
       24      Home Owner     Employed            -1         19         Yes       449.61       0
```

Score the new data and see how points are assigned for missing CustAge and ResStatus, because we have an explicit bin with points for <missing>. However, for EmpStatus and CustIncome the score function sets the points to NaN.

The validation results are unreliable, the scores with NaN values are kept (see the validation table ValTable below), but it is unclear what impact these NaN values have in the validation statistics (ValStats). This is a very small validation data set, but NaN scores could still influence the validation results on a larger data set.

```
[Scores,Points] = score(sc,tdata);
disp(Scores)
```

```
  481.2231
  520.8353
       NaN
       NaN
  551.7922
  487.9588
       NaN
       NaN
```

```
disp(Points)
```

```
    CustAge    ResStatus    EmpStatus    CustIncome    TmWBank    OtherCC    AMBalance
    _______    _________    _________    __________    _______    _______    _________

    64.984      62.138       58.807        67.893      61.858     75.622      89.922  
     78.86      74.125       58.807        82.439      61.061     75.622      89.922  
     96.76      73.248          NaN        96.969      51.132     50.914      89.922  
    69.636      90.828       58.807           NaN      61.858     50.914      89.922  
    64.984      90.828       86.937        82.439      61.061     75.622      89.922  
    56.282      74.125       86.937        70.107      61.858     75.622      63.028  
    60.012      62.138          NaN        67.893      61.858     75.622      63.028  
    54.062      73.248       86.937           NaN      61.061     75.622      89.922  
```

```
[ValStats,ValTable] = validatemodel(sc,tdata);
disp(ValStats)
```

```
            Measure                Value 
    ________________________    _______

    {'Accuracy Ratio'      }    0.16667
    {'Area under ROC curve'}    0.58333
    {'KS statistic'        }        0.5
    {'KS score'            }     481.22
```

```
disp(ValTable)
```

```
    Scores    ProbDefault    TrueBads    FalseBads    TrueGoods    FalseGoods    Sensitivity
    ______    ___________    ________    _________    _________    __________    ___________

       NaN            NaN       0            1            5             2                0  
       NaN            NaN       0            2            4             2                0  
       NaN            NaN       1            2            4             1              0.5  
       NaN            NaN       1            3            3             1              0.5  
    481.22        0.39345       2            3            3             0                1  
    487.96         0.3714       2            4            2             0                1  
    520.84         0.2725       2            5            1             0                1  
    551.79        0.19605       2            6            0             0                1  
```

Use the name-value argument `'Missing'` in `formatpoints` to choose how to assign points to missing values for predictors that do not have an explicit `<missing>` bin. In this example, use the `'MinPoints'` option for the `'Missing'` argument. The minimum points for `EmpStatus` in the scorecard displayed above are `58.8072` and for `CustIncome` the minimum points are `29.3753`.

The validation results are no longer influenced by `NaN` values, since all rows now have a score.

```
sc = formatpoints(sc,'Missing','MinPoints');
[Scores,Points] = score(sc,tdata);
disp(Scores)
```

```
  481.2231
  520.8353
  517.7532
  451.3405
  551.7922
  487.9588
  449.3577
  470.2267
```

```
disp(Points)
```

```
    CustAge    ResStatus    EmpStatus    CustIncome    TmWBank    OtherCC    AMBalance
    _______    _________    _________    __________    _______    _______    _________

    64.984      62.138       58.807        67.893      61.858     75.622      89.922  
     78.86      74.125       58.807        82.439      61.061     75.622      89.922  
     96.76      73.248       58.807        96.969      51.132     50.914      89.922  
    69.636      90.828       58.807        29.375      61.858     50.914      89.922  
    64.984      90.828       86.937        82.439      61.061     75.622      89.922  
    56.282      74.125       86.937        70.107      61.858     75.622      63.028  
    60.012      62.138       58.807        67.893      61.858     75.622      63.028  
    54.062      73.248       86.937        29.375      61.061     75.622      89.922  
```

```
[ValStats,ValTable] = validatemodel(sc,tdata);
disp(ValStats)
```

```
            Measure              Value 
    ________________________    _______

    {'Accuracy Ratio'      }    0.66667
    {'Area under ROC curve'}    0.83333
    {'KS statistic'        }    0.66667
    {'KS score'            }     481.22
```

```
disp(ValTable)
```

```
    Scores    ProbDefault    TrueBads    FalseBads    TrueGoods    FalseGoods    Sensitivity
    ______    ___________    ________    _________    _________    __________    ___________

    449.36      0.50223         1            0            6            1             0.5    
    451.34      0.49535         1            1            5            1             0.5    
    470.23      0.43036         1            2            4            1             0.5    
    481.22      0.39345         2            2            4            0               1    
    487.96       0.3714         2            3            3            0               1    
    517.75      0.28105         2            4            2            0               1    
    520.84       0.2725         2            5            1            0               1    
    551.79      0.19605         2            6            0            0               1    
```

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 15-1868.

Data Types: `table`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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 the comma-separated pair consisting of `'AnalysisLevel'` and a 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 the comma-separated pair consisting of `'Plot'` and a 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 15-1867.
- `'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 15-1867.
- `'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 15-1867.

> **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.

More About

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 also the credit scorecard scores themselves depend on the observation weights of the training data. For more information, see “Using fitmodel with Weights” on page 15-1722 and “Credit Scorecard Modeling Using Observation Weights” on page 8-54.

Version History

Introduced in R2015a

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

`creditscorecard` | `bininfo` | `predictorinfo` | `modifypredictor` | `modifybins` | `bindata` | `plotbins` | `fitmodel` | `displaypoints` | `formatpoints` | `score` | `setmodel` | `probdefault` | `table`

Topics

“Case Study for Credit Scorecard Analysis” on page 8-70
“Credit Scorecard Modeling with Missing Values” on page 8-56
“Troubleshooting Credit Scorecard Results” on page 8-63
“Credit Scorecard Modeling Workflow” on page 8-51
“About Credit Scorecards” on page 8-47
“Credit Scorecard Modeling Using Observation Weights” on page 8-54

fillmissing

Replace missing values for credit scorecard predictors

Syntax

```
sc = fillmissing(sc,PredictorNames,Statistics)
sc = fillmissing( ___ ,ConstantValue)
```

Description

`sc = fillmissing(sc,PredictorNames,Statistics)` replaces missing values of the predictor `PredictorNames` with values defined by `Statistics` and returns an updated credit scorecard object (`sc`). Standard missing data is defined as follows:

- `NaN` for numeric arrays
- `<undefined>` for categorical arrays

Note If you run `fillmissing` after binning a predictor, the existing cutpoints and bin edges are preserved and the "Good" and "Bad" counts from the `<missing>` bin are added to the corresponding bin.

`sc = fillmissing( ___ ,ConstantValue)` uses arguments from the previous syntax and a value for a `ConstantValue` to replace missing values.

Examples

Fill Missing Data in a `creditscorecard` Object

This example shows how to use `fillmissing` to replace missing values in the `CustAge` and `ResStatus` predictors with user-defined values. For additional information on alternative approaches for "treating" missing data, see “Credit Scorecard Modeling with Missing Values” on page 8-56.

Load the credit scorecard data and use `dataMissing` for the training data.

```
load CreditCardData.mat
disp(head(dataMissing));
```

```
    CustID    CustAge    TmAtAddress     ResStatus     EmpStatus    CustIncome    TmWBank    Othe
    ______    _______    ___________    ___________    _________    __________    _______    ____

      1          53          62         <undefined>    Unknown        50000         55         Ye
      2          61          22         Home Owner     Employed       52000         25         Ye
      3          47          30         Tenant         Employed       37000         61         No
      4         NaN          75         Home Owner     Employed       53000         20         Ye
      5          68          56         Home Owner     Employed       53000         14         Ye
      6          65          13         Home Owner     Employed       48000         59         Ye
      7          34          32         Home Owner     Unknown        32000         26         Ye
      8          50          57         Other          Employed       51000         33         No
```

Create a `creditscorecard` object with `'BinMissingData'` set to `true`.

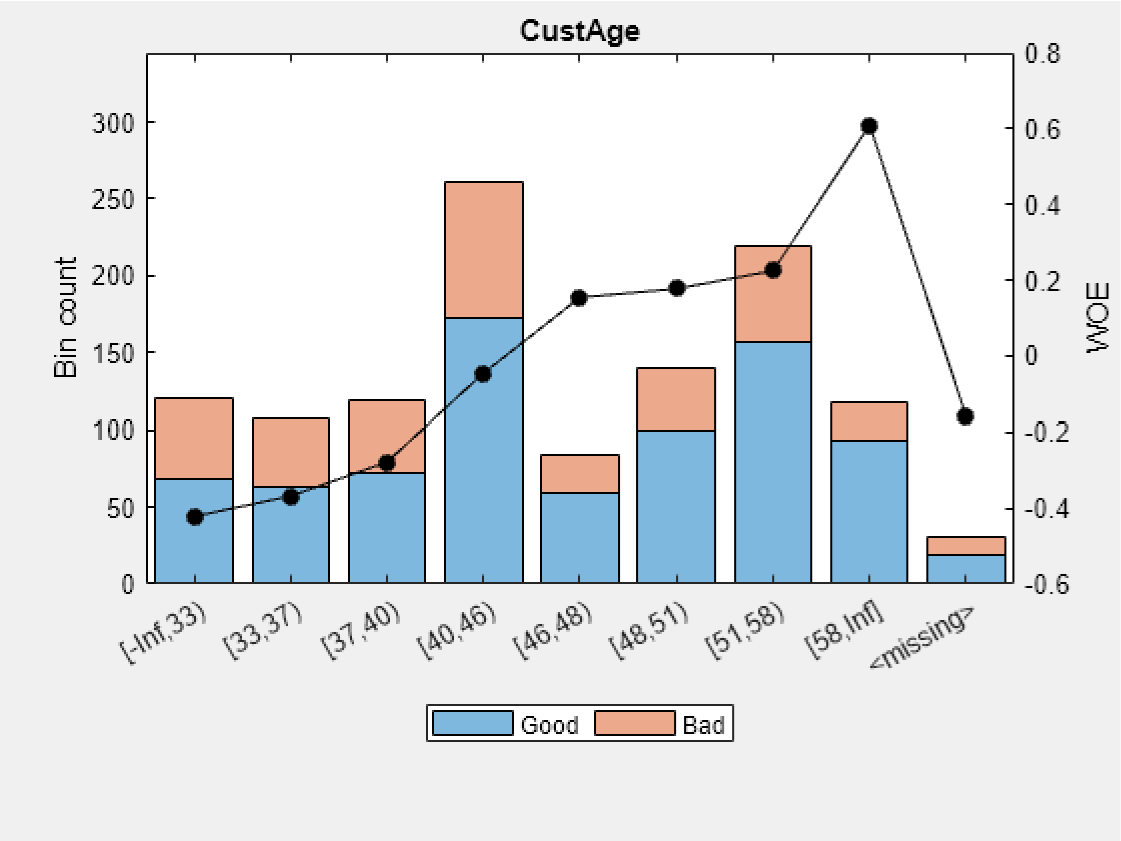
```
sc = creditscorecard(dataMissing,'BinMissingData',true);
sc = autobinning(sc);
```

Use `bininfo` and `plotbins` to display the `CustAge` and `ResStatus` predictors with missing data.

```
bininfo(sc,'CustAge')
```

```
ans=10×6 table
         Bin           Good    Bad     Odds        WOE       InfoValue 
    _____________      ____    ___    ______    _________    __________

    {'[-Inf,33)'}       69     52     1.3269     -0.42156      0.018993
    {'[33,37)'  }       63     45        1.4     -0.36795      0.012839
    {'[37,40)'  }       72     47     1.5319      -0.2779     0.0079824
    {'[40,46)'  }      172     89     1.9326     -0.04556     0.0004549
    {'[46,48)'  }       59     25       2.36      0.15424     0.0016199
    {'[48,51)'  }       99     41     2.4146      0.17713     0.0035449
    {'[51,58)'  }      157     62     2.5323      0.22469     0.0088407
    {'[58,Inf]' }       93     25       3.72      0.60931      0.032198
    {'<missing>'}       19     11     1.7273     -0.15787    0.00063885
    {'Totals'   }      803    397     2.0227          NaN      0.087112
```

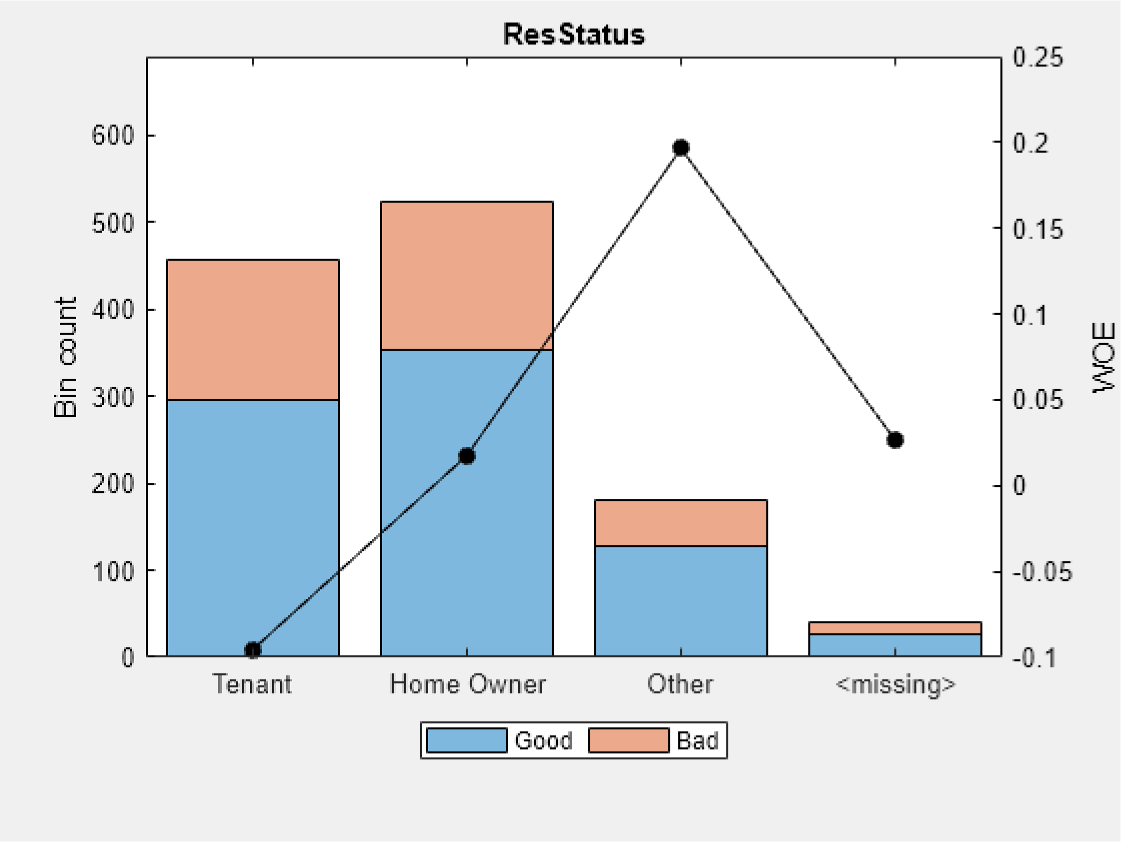
```
plotbins(sc,'CustAge');
```



```
bininfo(sc,'ResStatus')
```

```
ans=5×6 table
          Bin            Good    Bad     Odds        WOE       InfoValue 
    _______________    ____    ___    ______    _________    __________

    {'Tenant'     }    296     161    1.8385    -0.095463     0.0035249
    {'Home Owner'}     352     171    2.0585     0.017549    0.00013382
    {'Other'      }    128      52    2.4615      0.19637     0.0055808
    {'<missing>' }      27      13    2.0769     0.026469    2.3248e-05
    {'Totals'     }    803     397    2.0227          NaN     0.0092627
```

```
plotbins(sc,'ResStatus');
```



Use `fillmissing` to replace `NaN` values in `CustAge` with the median value and to replace the `<missing>` values in `ResStatus` with `'Tenant'`. Use `predictorinfo` to verify the filled values.

```
sc = fillmissing(sc,{'CustAge'},'median');
sc = fillmissing(sc,{'ResStatus'},'constant','Tenant');
predictorinfo(sc,'CustAge')
```

```
ans=1×4 table
                PredictorType           LatestBinning            LatestFillMissingType    LatestFillMi
               _____________    ________________________    _____________________    ____________

    CustAge    {'Numeric'}    {'Automatic / Monotone'}         {'Median'}                  {[4
```

```
predictorinfo(sc,'ResStatus')
```

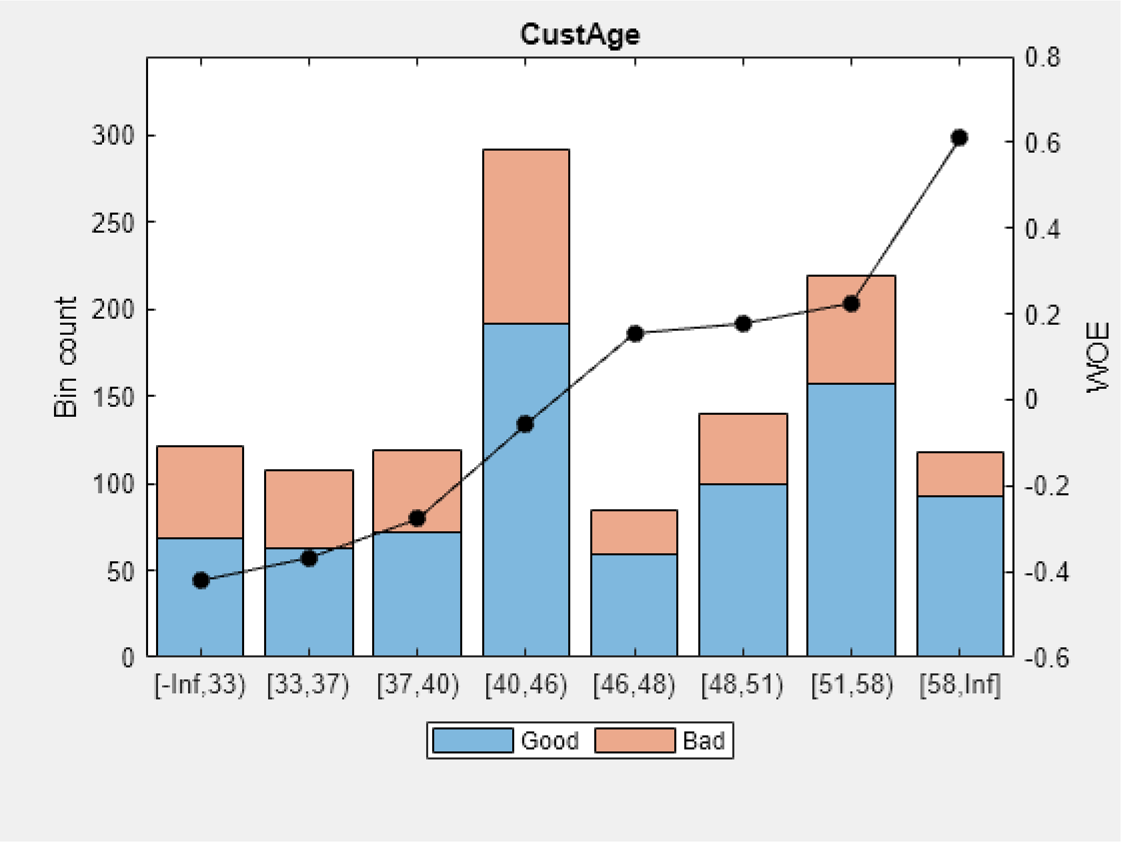
```
ans=1×5 table
                 PredictorType     Ordinal        LatestBinning          LatestFillMissingType
                _______________    _______    ________________________    _____________________

    ResStatus   {'Categorical'}     false     {'Automatic / Monotone'}        {'Constant'}
```

Use `bininfo` and `plotbins` to display the `CustAge` and `ResStatus` predictors to verify that the missing data has been replaced with the values defined by `fillmissing`.

```
bininfo(sc,'CustAge')
```

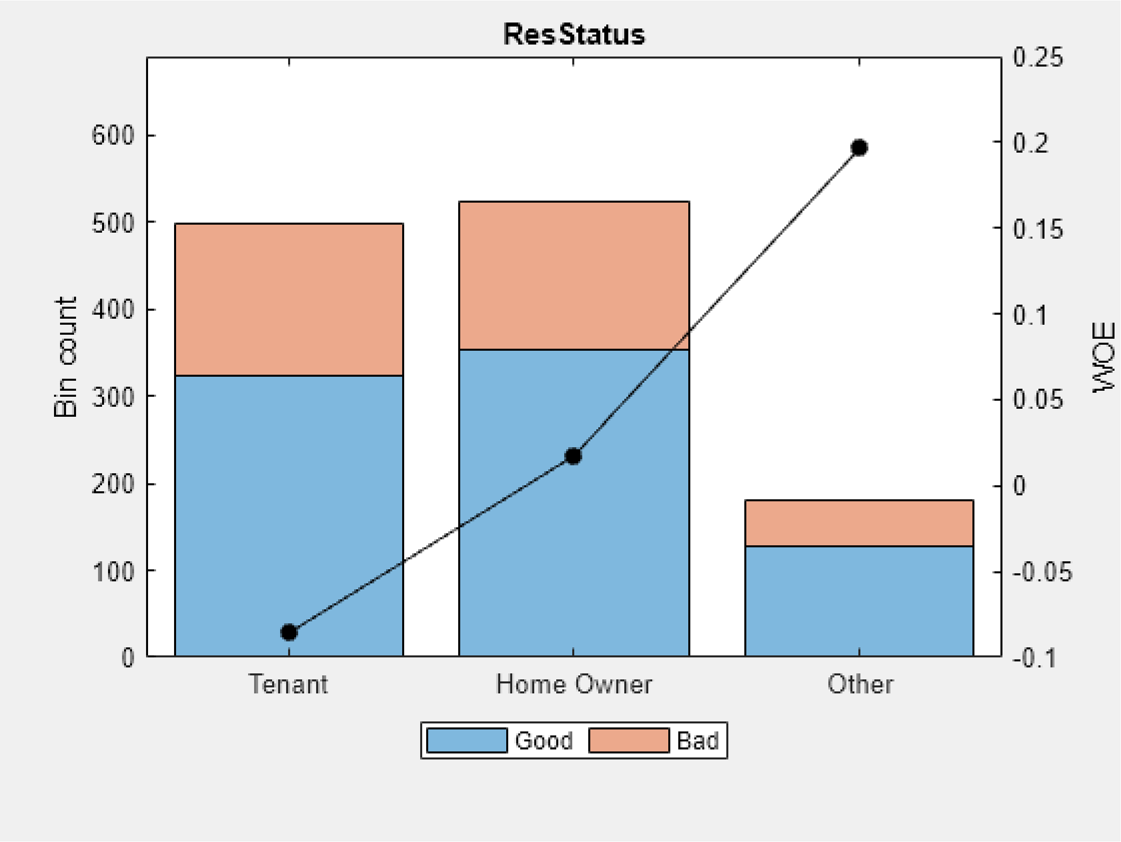
```
ans=9×6 table
         Bin          Good    Bad     Odds        WOE       InfoValue
    _____________    ____    ___    ______    _________    _________

    {'[-Inf,33)'}     69      52    1.3269     -0.42156     0.018993
    {'[33,37)'  }     63      45       1.4     -0.36795     0.012839
    {'[37,40)'  }     72      47    1.5319      -0.2779    0.0079824
    {'[40,46)'  }    191     100      1.91    -0.057315    0.0008042
    {'[46,48)'  }     59      25      2.36      0.15424    0.0016199
    {'[48,51)'  }     99      41    2.4146      0.17713    0.0035449
    {'[51,58)'  }    157      62    2.5323      0.22469    0.0088407
    {'[58,Inf]' }     93      25      3.72      0.60931     0.032198
    {'Totals'   }    803     397    2.0227          NaN     0.086822
```

```
plotbins(sc,'CustAge');
```



```
bininfo(sc,'ResStatus')

ans=4×6 table
          Bin           Good    Bad     Odds        WOE         InfoValue 
    ________________    ____    ___    ______    _________    __________

    {'Tenant'      }    323     174    1.8563    -0.085821     0.0030935
    {'Home Owner'}      352     171    2.0585     0.017549    0.00013382
    {'Other'       }    128      52    2.4615      0.19637     0.0055808
    {'Totals'      }    803     397    2.0227          NaN     0.0088081
```

```
plotbins(sc,'ResStatus');
```



Use `fitmodel` and then run `formatpoints`, `displaypoints`, and `score`.

```
sc = fitmodel(sc,'Display','off');
sc = formatpoints(sc,'WorstAndBest',[300 800]);
t = displaypoints(sc)

t=31×3 table
       Predictors              Bin             Points
    ______________    _________________    ______

    {'CustAge'   }    {'[-Inf,33)'    }    72.565
    {'CustAge'   }    {'[33,37)'      }    76.588
    {'CustAge'   }    {'[37,40)'      }    83.346
    {'CustAge'   }    {'[40,46)'      }    99.902
    {'CustAge'   }    {'[46,48)'      }    115.78
    {'CustAge'   }    {'[48,51)'      }     117.5
    {'CustAge'   }    {'[51,58)'      }    121.07
    {'CustAge'   }    {'[58,Inf]'     }    149.93
    {'CustAge'   }    {'<missing>'    }    99.902
    {'EmpStatus' }    {'Unknown'      }     79.64
    {'EmpStatus' }    {'Employed'     }    133.98
    {'EmpStatus' }    {'<missing>'    }       NaN
    {'CustIncome'}    {'[-Inf,29000)' }    21.926
    {'CustIncome'}    {'[29000,33000)'}    73.949
    {'CustIncome'}    {'[33000,35000)'}    97.117
    {'CustIncome'}    {'[35000,40000)'}    101.44
      ⋮
```

When a validation data set has missing values and you use `fillmissing` with the training dataset, the missing values in the validation data set are assigned the same points as the corresponding bins containing the filled values.

As the table shows, the `'<missing>'` bin for the `CustAge` predictor is assigned the same points as the `'[40,46)'` bin because the missing data is filled with the median value `45`.

The points assigned to the `'<missing>'` bin for the `EmpStatus` predictor are `NaN` because `fillmissing` is not used for that predictor. The assigned points are decided by the default `'NoScore'` for the `'Missing'` name-value pair argument in `formatpoints`.

Create a test validation data set (`tdata`) and add missing values.

```
tdata = data(1:10,:);
tdata.CustAge(1) = NaN;
tdata.ResStatus(2) = '<undefined>';
[scr,pts] = score(sc,tdata)
```

```
scr = 10×1

  566.7335
  611.2547
  584.5130
  628.7876
  609.7148
  671.1048
  403.6413
  551.9461
  575.9874
  524.4789


pts=10×5 table
    CustAge    EmpStatus    CustIncome    TmWBank    AMBalance
    _______    _________    __________    _______    _________

    99.902       79.64        153.88      145.38      87.933 
    149.93      133.98        153.88      85.531      87.933 
    115.78      133.98        101.44      145.38      87.933 
     117.5      133.98        153.88      83.991      139.44 
    149.93      133.98        153.88      83.991      87.933 
    149.93      133.98        153.88      145.38      87.933 
    76.588       79.64       73.949      85.531      87.933
     117.5      133.98       153.88      85.531       61.06
     117.5       79.64       153.88      85.531      139.44
     117.5       79.64       153.88      85.531      87.933
```

Handling of Missing Values in Validation Data Sets

This example shows different possibilities for handling missing data in validation data.

When scoring data from a validation data set, you have several options. If you choose to do nothing, the points assigned to the missing data are `NaN`, which comes from the default `'NoScore'` for the `'Missing'` name-value pair argument in `formatpoints`.

If you want to score missing values of all the predictors with one consistent metric, you can use the options `'ZeroWOE'`, `'MinPoints'`, or `'MaxPoints'` for the `'Missing'` name-value pair argument in `formatpoints`.

```
load CreditCardData.mat
sc = creditscorecard(data);
predictorinfo(sc,'CustAge')
```

```
ans=1×4 table
               PredictorType      LatestBinning       LatestFillMissingType    LatestFillMissingVa
               _____________    __________________    _____________________    ___________________

    CustAge     {'Numeric'}     {'Original Data'}         {'Original'}               {0x0 double}
```

```
predictorinfo(sc,'ResStatus')
```

```
ans=1×5 table
                 PredictorType      Ordinal      LatestBinning       LatestFillMissingType    Late
                _______________    _______    __________________    _____________________    ____

    ResStatus   {'Categorical'}     false     {'Original Data'}         {'Original'}
```

```
sc = autobinning(sc);
sc = fitmodel(sc,'display','off');

displaypoints(sc)
```

```
ans=37×3 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
    {'CustAge'   }    {'<missing>'   }          NaN
    {'ResStatus' }    {'Tenant'      }    -0.031252
    {'ResStatus' }    {'Home Owner'  }      0.12696
    {'ResStatus' }    {'Other'       }      0.37641
    {'ResStatus' }    {'<missing>'   }          NaN
    {'EmpStatus' }    {'Unknown'     }    -0.076317
    {'EmpStatus' }    {'Employed'    }      0.31449
    {'EmpStatus' }    {'<missing>'   }          NaN
    {'CustIncome'}    {'[-Inf,29000)'}     -0.45716
      ⋮
```

```
sc = formatpoints(sc,'Missing','minpoints','WorstAndBestScores',[300 850]);
displaypoints(sc)
```

```
ans=37×3 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
    {'CustAge'   }    {'<missing>'   }    46.396
    {'ResStatus' }    {'Tenant'      }    62.421
    {'ResStatus' }    {'Home Owner'  }    82.276
    {'ResStatus' }    {'Other'       }    113.58
    {'ResStatus' }    {'<missing>'   }    62.421
    {'EmpStatus' }    {'Unknown'     }    56.765
    {'EmpStatus' }    {'Employed'    }    105.81
    {'EmpStatus' }    {'<missing>'   }    56.765
    {'CustIncome'}    {'[-Inf,29000)'}    8.9706
      ⋮
```

The value of `-32.5389` for the `<missing>` bin of `'CustAge'` comes from the `'minPoints'` argument for `formatpoints`.

```
[scr,pts] = score(sc,dataMissing(1:5,:))
```

```
scr = 5×1

  602.0394
  648.1988
  560.5569
  613.5595
  646.8109
```

```
pts=5×7 table
    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  
    46.396      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
```

Alternatively, you can score missing data for each individual predictor with a different statistic based on that predictor's information. To do so, use `fillmissing` for a `creditscorecard` object.

```
load CreditCardData.mat
sc = creditscorecard(data);
sc = fillmissing(sc,'CustAge','constant',35);
predictorinfo(sc,'CustAge')
```

```
ans=1×4 table
               PredictorType      LatestBinning       LatestFillMissingType    LatestFillMissingVa
               _____________    __________________    _____________________    ___________________

    CustAge     {'Numeric'}     {'Original Data'}         {'Constant'}                  {[35]}
```

```
sc = fillmissing(sc,'ResStatus','Mode');
predictorinfo(sc,'ResStatus')
```

```
ans=1×5 table
                 PredictorType      Ordinal      LatestBinning       LatestFillMissingType    Late
                _______________    _______    __________________    _____________________    ____

    ResStatus    {'Categorical'}     false     {'Original Data'}            {'Mode'}
```

```
sc = autobinning(sc);
sc = fitmodel(sc,'display','off');
sc = formatpoints(sc,'Missing','minpoints','WorstAndBestScores',[300 850]);

displaypoints(sc)
```

```
ans=37×3 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
    {'CustAge'   }    {'<missing>'   }    48.727
    {'ResStatus' }    {'Tenant'      }    62.421
    {'ResStatus' }    {'Home Owner'  }    82.276
    {'ResStatus' }    {'Other'       }    113.58
    {'ResStatus' }    {'<missing>'   }    82.276
    {'EmpStatus' }    {'Unknown'     }    56.765
    {'EmpStatus' }    {'Employed'    }    105.81
    {'EmpStatus' }    {'<missing>'   }    56.765
    {'CustIncome'}    {'[-Inf,29000)'}    8.9706
      ⋮
```

The value of `<missing>` for `'CustAge'` comes from the fill value of `35` even though the training data has no missing values.

```
disp(dataMissing(1:5,:));
```

```
    CustID    CustAge    TmAtAddress     ResStatus     EmpStatus    CustIncome    TmWBank    Othe
    ______    _______    ___________    ___________    _________    __________    _______    ____

      1         53           62         <undefined>    Unknown        50000         55         Ye
      2         61           22         Home Owner     Employed       52000         25         Ye
      3         47           30         Tenant         Employed       37000         61         No
      4        NaN           75         Home Owner     Employed       53000         20         Ye
      5         68           56         Home Owner     Employed       53000         14         Ye
```

```
[scr,pts] = score(sc,dataMissing(1:5,:))
```

```
scr = 5×1

  621.8943
  648.1988
  560.5569
  615.8904
  646.8109


pts=5×7 table
    CustAge    ResStatus    EmpStatus    CustIncome    TmWBank    OtherCC    AMBalance
    _______    _________    _________    __________    _______    _______    _________

    95.256      82.276       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
    48.727      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
```

Input Arguments

`sc` — Credit scorecard model

`creditscorecard` object

Credit scorecard model, specified as a `creditscorecard` object.

`PredictorNames` — Name of `creditscorecard` predictor whose missing data is filled

character vector | string | cell array of character vectors | string array

Name of `creditscorecard` predictor to fill missing data for, specified as a scalar character vector, scalar string, cell array of character vectors, or string array.

Data Types: `char` | `string` | `cell`

`Statistics` — Statistic to use to fill missing data for predictors

character vector with a value of `'mean'`, `'median'`, `'mode'`, `'original'`, or `'constant'` | string with a value of `"mean"`, `"median"`, `"mode"`, `"original"`, or `"constant"`

Statistic to use to fill missing data for the predictors, specified as a character vector or string with one of the following values.

- `'mean'` — Replace missing data with the average or mean value. The option is valid only for numeric data. The `'mean'` calculates the weighted mean of the predictor by referring to the predictor column and the `Weights` column from the `creditscorecard` object. For more information, see “Weighted Mean” on page 15-1879.

- `'median'` — Replace missing data with the median value. Valid for numeric and ordinal data. The `'median'` calculates the weighted median of the predictor by referring to the predictor column and the `Weights` column from the `creditscorecard` object. For more information, see “Weighted Median” on page 15-1880.
- `'mode'` — Replace missing data with the mode. Valid for numeric and both nominal and ordinal categorical data. The `'mode'` calculates the weighted `mode` of the predictor by referring to the predictor column and the `Weights` column from the `creditscorecard` object. For more information, see “Weighted Mode” on page 15-1880.
- `'original'` — Set the missing data for numeric and categorical predictors back to its original value: `NaN` if numeric, `<undefined>` or `<missing>` if categorical.
- `'constant'` — Set the missing data for numeric and categorical predictors to a constant value that you specify in the optional argument for `ConstantValue`.

Data Types: `char` | `string`

`ConstantValue` — Value to fill missing entries in predictors specified in `PredictorNames`
`[ ]` (default) | numeric | character vector | string | cell array of character vectors | string array

(Optional) Value to fill missing entries in predictors specified in `PredictorNames`, specified as a numeric value, character vector, string, or cell array of character vectors.

Note You can use `ConstantValue` only if you set the `Statistics` argument to `'constant'`.

Data Types: `char` | `double` | `string` | `cell`

Output Arguments

`sc` — Updated `creditscorecard`
`creditscorecard` object

Updated `creditscorecard` object, returned as an object.

More About

Weighted Mean

The weighted mean is similar to an ordinary mean except that instead of each of the data points contributing equally to the final average, some data points contribute more than others.

The weighted mean for a nonempty finite multiset of data (x) with corresponding nonnegative weights (w) is

$$\bar{x} = \frac{\sum_{i=1}^{n} w_i x_i}{\sum_{i=1}^{n} w_i}$$

Weighted Median

The weighted median is the 50% weighted percentile, where the percentage in the total weight is counted instead of the total number.

For n distinct ordered elements (x) positive weights (w) such that $\sum_{i=1}^{n} w_i = 1$, the weighted median is the element x_k:

$$\sum_{i=1}^{k-1} w_i \leq \frac{1}{2} \text{ and } \sum_{i=k+1}^{n} w_i \leq \frac{1}{2}$$

In the case where the respective weights of both elements border the midpoint of the set of weights without encapsulating it, each element defines a partition equal to 1/2. These elements are referred to as the lower weighted median and upper weighted median. The weighted median is chosen based on which element keeps the partitions most equal. This median is always the weighted median with the lowest weight. In the event that the upper and lower weighted medians are equal, the lower weighted median is accepted.

Weighted Mode

The weighted mode of a set of weighted data values is the value that appears most often.

The mode of a sample is the element that occurs most often in the collection. For example, the mode of the sample [1, 3, 6, 6, 6, 6, 7, 7, 12, 12, 17] is 6.

Version History

Introduced in R2020a

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

`creditscorecard` | `bininfo` | `predictorinfo` | `modifypredictor` | `modifybins` | `bindata` | `plotbins` | `fitmodel` | `displaypoints` | `formatpoints` | `score` | `setmodel` | `probdefault` | `validatemodel` | `table`

Topics

"Case Study for Credit Scorecard Analysis" on page 8-70
"Credit Scorecard Modeling with Missing Values" on page 8-56
"Troubleshooting Credit Scorecard Results" on page 8-63
"Credit Scorecard Modeling Workflow" on page 8-51
"About Credit Scorecards" on page 8-47
"Credit Scorecard Modeling Using Observation Weights" on page 8-54

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. Use `screenpredictors` from Risk Management Toolbox to pare down a potentially large set of predictors to a subset that is most predictive of the credit score card response variable. Use this subset of predictors when creating the `creditscorecard` object.
2. Create a `creditscorecard` object (see “Create creditscorecard” on page 15-1881 and “Properties” on page 15-1884).
3. Bin the data using `autobinning`.
4. Fit a logistic regression model using `fitmodel` or `fitConstrainedModel`.
5. Review and format the credit scorecard points using `displaypoints` and `formatpoints`. At this point in the workflow, if you have a license for Risk Management Toolbox, you have the option to create a `compactCreditScorecard` object (`csc`) using the `compact` function. You can then use the following functions `displaypoints`, `score`, and `probdefault` from the Risk Management Toolbox with the `csc` object.
6. Score the data using `score`.
7. Calculate the probabilities of default for the data using `probdefault`.
8. Validate the quality of the credit scorecard model using `validatemodel`.

For more detailed information on this workflow, see “Credit Scorecard Modeling Workflow” on page 8-51.

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 15-1884 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', {'CustAge','CustIncome'},'WeightsVar','RowWeights','BinMissingData',true)`. 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 15-0 property.

Data Types: `table`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `sc = creditscorecard(data,'GoodLabel',0,'IDVar','CustAge','ResponseVar','status','PredictorVars', {'CustID','CustIncome'},'WeightsVar','RowWeights','BinMissingData',true)`

`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 the comma-separated pair consisting of `'GoodLabel'` and a numeric scalar, logical, or character vector. The `GoodLabel` name-value pair argument sets the GoodLabel on page 15-0 property.

When specifying `GoodLabel`, follow these guidelines.

If Response Variable 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 8-54.

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 the comma-separated pair consisting of 'IDVar' and 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 15-0 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 the comma-separated pair consisting of 'ResponseVar' and a character vector. The response variable data must be binary. The ResponseVar name-value pair argument sets the ResponseVar on page 15-0 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 the comma-separated pair consisting of 'WeightsVar' and 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 15-0 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 8-54.

Data Types: char

BinMissingData — Indicates if missing data is removed or displayed in a separate bin
false (missing data removed) (default) | logical with value true or false

Indicates if missing data is removed or displayed in a separate bin, specified as the comma-separated pair consisting of 'BinMissingdata' and a logical scalar with a value of true or false. If BinMissingData is true, the missing data for a predictor is displayed in a separate bin labeled <missing>.

Data Types: logical

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 Variable 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. 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`

`BinMissingData` — Indicates if missing data is removed or displayed in a separate bin

`false` (missing data removed) (default) | logical with value `true` or `false`

Indicates if missing data is removed or displayed in a separate bin, specified as the comma-separated pair consisting of `'BinMissingdata'` and a logical scalar with a value of `true` or `false`. If `BinMissingData` is `true`, the missing data for a predictor is displayed in a separate bin labeled `<missing>`. For more information on working with missing data, see “Credit Scorecard Modeling with Missing Values” on page 8-56.

Data Types: `logical`

`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

`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
`BinMissingData`	Yes	No	False by default, 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
fillmissing	Replace missing values for 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
fitConstrainedModel	Fit logistic regression model to Weight of Evidence (WOE) data subject to constraints on model coefficients
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
compact	Create compact credit scorecard

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: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIn
        NumericPredictors: {'CustID'  'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBalan
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
           BinMissingData: 0
                    IDVar: ''
            PredictorVars: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIn
                     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'
               WeightsVar: 'RowWeights'
                 VarNames: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustI
        NumericPredictors: {'CustID'  'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBala
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
           BinMissingData: 0
                    IDVar: ''
            PredictorVars: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustI
                     Data: [1200x12 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: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustI
        NumericPredictors: {'CustID'  'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBala
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
           BinMissingData: 0
                    IDVar: ''
            PredictorVars: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustI
                     Data: [1200x11 table]
```

To display the creditscorecard object properties, use dot notation.

```
sc.PredictorVars
```

```
ans = 1x10 cell
    {'CustID'}    {'CustAge'}    {'TmAtAddress'}    {'ResStatus'}    {'EmpStatus'}    {'CustInco
```

```
sc.VarNames
```

```
ans = 1x11 cell
    {'CustID'}    {'CustAge'}    {'TmAtAddress'}    {'ResStatus'}    {'EmpStatus'}    {'CustInco
```

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: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustI
        NumericPredictors: {'CustID'  'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBala
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
           BinMissingData: 0
                    IDVar: ''
            PredictorVars: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustI
                     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: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustI
        NumericPredictors: {'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBalance'  'Uti
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
           BinMissingData: 0
                    IDVar: 'CustID'
            PredictorVars: {'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIncome'  'T
                     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: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustI
        NumericPredictors: {'CustID'  'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBala
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
           BinMissingData: 0
                    IDVar: ''
            PredictorVars: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustI
                     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
    {'CustID'}    {'CustAge'}    {'TmAtAddress'}    {'ResStatus'}    {'EmpStatus'}    {'CustInco
```

```
sc.VarNames
```

```
ans = 1x11 cell
    {'CustID'}    {'CustAge'}    {'TmAtAddress'}    {'ResStatus'}    {'EmpStatus'}    {'CustInco
```

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: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustI
        NumericPredictors: {'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBalance'  'Uti
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
```

```
           BinMissingData: 0
                    IDVar: 'CustID'
            PredictorVars: {'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIncome'  'Tr
                     Data: [1200x11 table]
```

```
sc.PredictorVars = {'CustIncome','ResStatus','AMBalance'}
```

```
sc = 
  creditscorecard with properties:

                GoodLabel: 0
              ResponseVar: 'status'
               WeightsVar: ''
                 VarNames: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustI
        NumericPredictors: {'CustIncome'  'AMBalance'}
    CategoricalPredictors: {'ResStatus'}
           BinMissingData: 0
                    IDVar: 'CustID'
            PredictorVars: {'ResStatus'  'CustIncome'  'AMBalance'}
                     Data: [1200x11 table]
```

```
disp(sc)
```

```
  creditscorecard with properties:

                GoodLabel: 0
              ResponseVar: 'status'
               WeightsVar: ''
                 VarNames: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustI
        NumericPredictors: {'CustIncome'  'AMBalance'}
    CategoricalPredictors: {'ResStatus'}
           BinMissingData: 0
                    IDVar: 'CustID'
            PredictorVars: {'ResStatus'  'CustIncome'  '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: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustI
        NumericPredictors: {'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBalance'  'Uti
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
```

```
          BinMissingData: 0
                   IDVar: 'CustID'
           PredictorVars: {'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIncome'  'Tı
                    Data: [1200x11 table]
```

`GoodLabel` and `ResponseVar` can only be set (enforced) when creating a `creditscorecard` object using `creditscorecard`.

Create a `creditscorecard` Object and Set the `'BinMissingData'` Property

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `dataMissing` with missing values.

```
load CreditCardData
head(dataMissing,5)
```

```
    CustID    CustAge    TmAtAddress     ResStatus      EmpStatus    CustIncome    TmWBank    Othe
    ______    _______    ___________    ___________    _________    __________    _______    ____

      1         53           62         <undefined>    Unknown        50000          55         Ye
      2         61           22         Home Owner     Employed       52000          25         Ye
      3         47           30         Tenant         Employed       37000          61         No
      4        NaN           75         Home Owner     Employed       53000          20         Ye
      5         68           56         Home Owner     Employed       53000          14         Ye
```

```
fprintf('Number of rows: %d\n',height(dataMissing))
```

```
Number of rows: 1200
```

```
fprintf('Number of missing values CustAge: %d\n',sum(ismissing(dataMissing.CustAge)))
```

```
Number of missing values CustAge: 30
```

```
fprintf('Number of missing values ResStatus: %d\n',sum(ismissing(dataMissing.ResStatus)))
```

```
Number of missing values ResStatus: 40
```

Use `creditscorecard` with the name-value argument `'BinMissingData'` set to `true` to bin the missing data in a separate bin.

```
sc = creditscorecard(dataMissing,'IDVar','CustID','BinMissingData',true);
sc = autobinning(sc);
disp(sc)
```

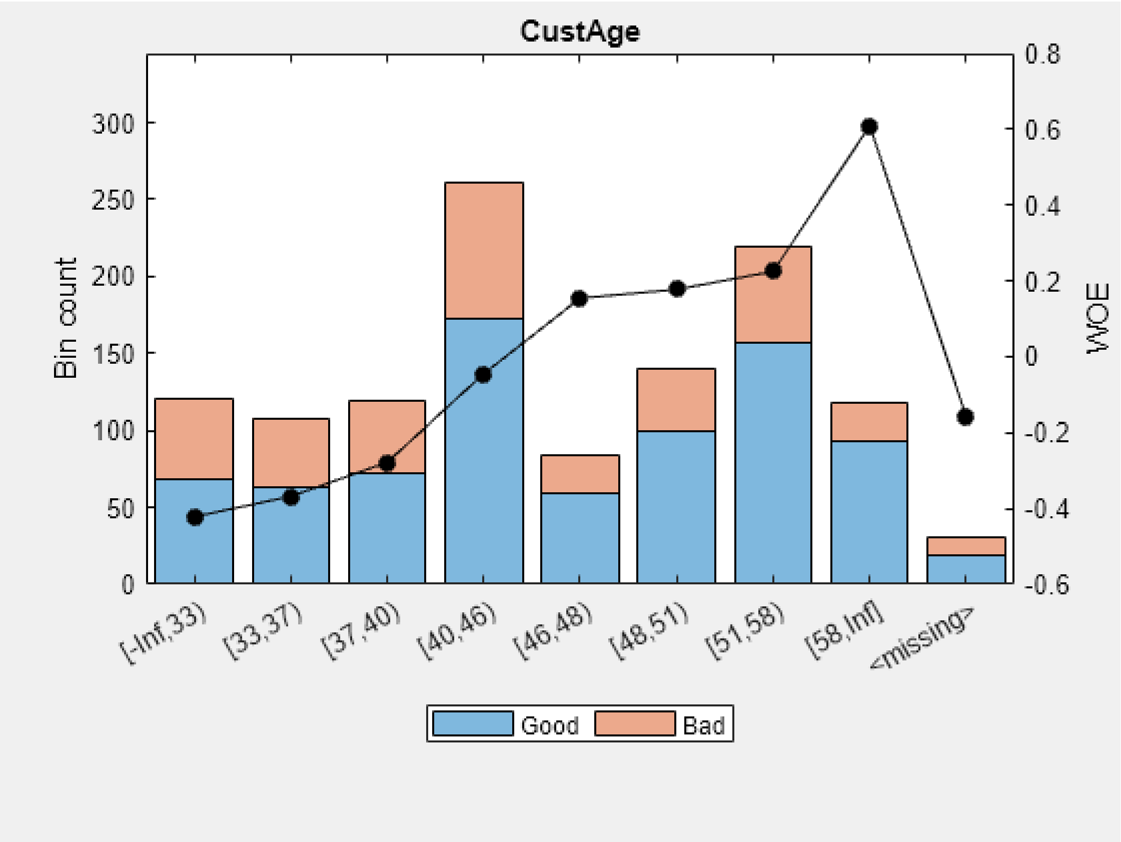
```
  creditscorecard with properties:

                GoodLabel: 0
              ResponseVar: 'status'
               WeightsVar: ''
                 VarNames: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIı
        NumericPredictors: {'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBalance'  'Uti
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
           BinMissingData: 1
                    IDVar: 'CustID'
            PredictorVars: {'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIncome'  'Tı
                     Data: [1200x11 table]
```

Display bin information for numeric data for `'CustAge'` that includes missing data in a separate bin labelled `<missing>`.

```
bi = bininfo(sc,'CustAge');
disp(bi)
```

```
         Bin           Good    Bad     Odds        WOE       InfoValue 
    ______________     ____    ___    ______    ________    ___________

    {'[-Inf,33)'}       69     52     1.3269    -0.42156       0.018993
    {'[33,37)'  }       63     45        1.4    -0.36795       0.012839
    {'[37,40)'  }       72     47     1.5319     -0.2779      0.0079824
    {'[40,46)'  }      172     89     1.9326    -0.04556      0.0004549
    {'[46,48)'  }       59     25       2.36     0.15424      0.0016199
    {'[48,51)'  }       99     41     2.4146     0.17713      0.0035449
    {'[51,58)'  }      157     62     2.5323     0.22469      0.0088407
    {'[58,Inf]' }       93     25       3.72     0.60931       0.032198
    {'<missing>'}       19     11     1.7273    -0.15787     0.00063885
    {'Totals'   }      803    397     2.0227         NaN       0.087112
```

```
plotbins(sc,'CustAge')
```

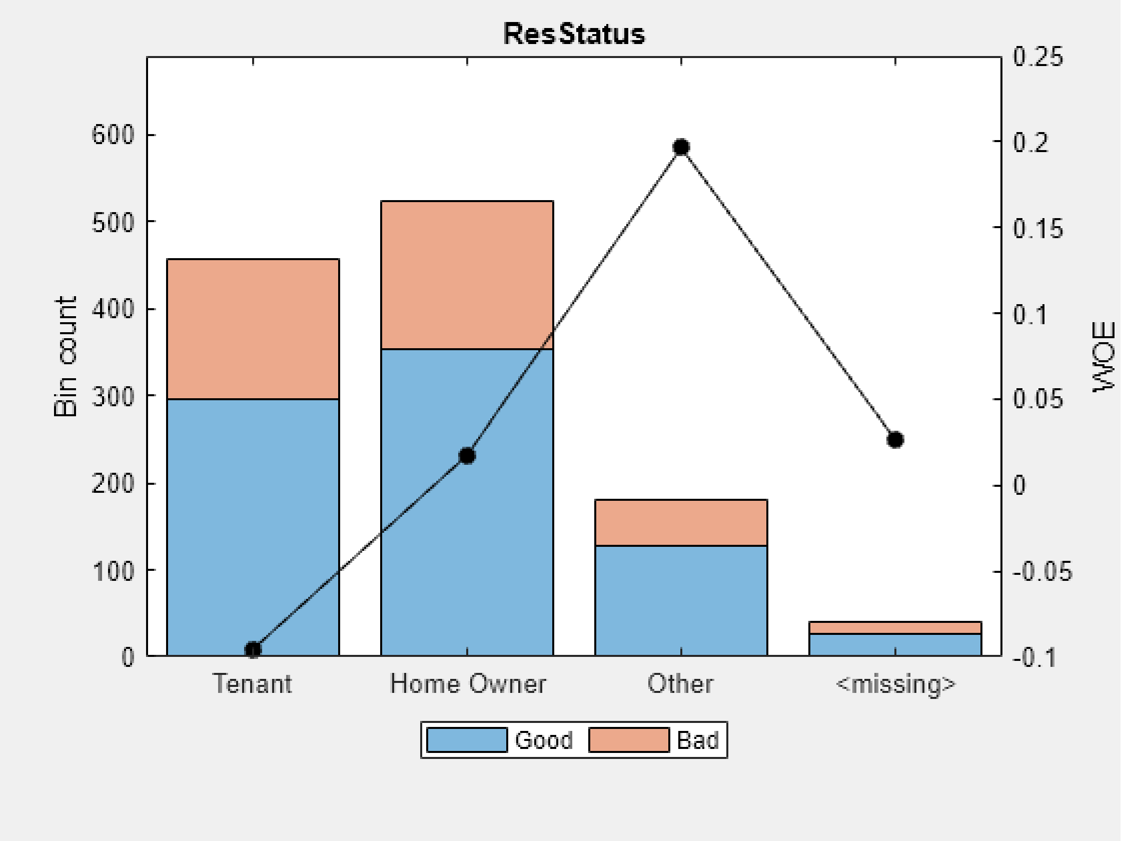


Display bin information for categorical data for `'ResStatus'` that includes missing data in a separate bin labelled `<missing>`.

```
bi = bininfo(sc,'ResStatus');
disp(bi)
```

```
         Bin            Good    Bad     Odds        WOE       InfoValue 
    ______________      ____    ___    ______    _________    __________

    {'Tenant'     }     296     161    1.8385    -0.095463     0.0035249
    {'Home Owner'}      352     171    2.0585     0.017549    0.00013382
    {'Other'      }     128      52    2.4615      0.19637     0.0055808
    {'<missing>' }       27      13    2.0769     0.026469    2.3248e-05
    {'Totals'     }     803     397    2.0227          NaN     0.0092627
```

```
plotbins(sc,'ResStatus')
```



Version History

Introduced in R2014b

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

`screenpredictors` | `autobinning` | `modifybins` | `bindata` | `bininfo` | `fillmissing` | `predictorinfo` | `modifypredictor` | `plotbins` | `fitmodel` | `fitConstrainedModel` | `displaypoints` | `formatpoints` | `score` | `setmodel` | `validatemodel` | `probdefault` | `table`

Apps

Binning Explorer

Topics

“Case Study for Credit Scorecard Analysis” on page 8-70
“Credit Scorecards with Constrained Logistic Regression Coefficients” on page 8-88
“Credit Scorecard Modeling with Missing Values” on page 8-56
“Credit Scoring Using Logistic Regression and Decision Trees” (Risk Management Toolbox)
“Use Reject Inference Techniques with Credit Scorecards” (Risk Management Toolbox)
“compactCreditScorecard Object Workflow” (Risk Management Toolbox)
“Troubleshooting Credit Scorecard Results” on page 8-63
“Bin Data to Create Credit Scorecards Using Binning Explorer” (Risk Management Toolbox)
“Explore Fairness Metrics for Credit Scoring Model” (Risk Management Toolbox)
“Bias Mitigation in Credit Scoring by Reweighting” (Risk Management Toolbox)
“Bias Mitigation in Credit Scoring by Disparate Impact Removal” (Risk Management Toolbox)
“Interpretability and Explainability for Credit Scoring” (Risk Management Toolbox)
“Credit Scorecard Modeling Workflow” on page 8-51
“About Credit Scorecards” on page 8-47
“Credit Scorecard Modeling Using Observation Weights” on page 8-54
“Overview of Binning Explorer” (Risk Management Toolbox)

External Websites

Credit Risk Modeling with MATLAB (53 min 10 sec)

addBusinessCalendar

Add business calendar awareness to timetables

Syntax

```
TT = addBusinessCalendar(TT)
TT = addBusinessCalendar( ___ ,Name=Value)
```

Description

`TT = addBusinessCalendar(TT)` adds business calendar awareness to an input timetable `TT` by setting a custom property for the output timetable `TT`.

`TT = addBusinessCalendar( ___ ,Name=Value)` sets additional options specified by one or more name-value arguments, using any of the input arguments in the previous syntax. For example, `TT = addBusinessCalendar(TT,Holidays=H)` replaces the default holidays stored in `Data_NYSE_Closures.mat` with the list of holidays `H`.

Examples

Add Business Calendar for Calculating Period-Over-Period Rolling Returns

This example shows how to add a business calendar when you calculate period-over-period (PoP) rolling differences for 5 years of simulated daily prices. For each date in timetable `TT`, the difference represents the PoP difference of the corresponding price compared to the price one period earlier.

Use `holidays` to indicate the holidays in `holidays.m` for the simulation period.

```
H = holidays(datetime(2014,1,1),datetime(2018,12,31));
```

Simulate daily prices for three assets.

```
t = (datetime(2014,1,1):caldays:datetime(2018,12,31))';
rng(200,"twister")
Price = 100 + 0.1*(0:numel(t) - 1)'.*cumsum(randn(numel(t),1)/100);
Price = round(Price*100)/100;
Price2 = round(Price*94)/100;
Price3 = round(Price*88)/100;

TT    = timetable(Price,Price2,Price3,RowTimes=t);
head(TT,15)
```

```
       Time        Price     Price2    Price3
    ___________    ______    ______    ______

    01-Jan-2014       100        94        88
    02-Jan-2014       100        94        88
    03-Jan-2014       100        94        88
    04-Jan-2014       100        94        88
    05-Jan-2014    100.01     94.01     88.01
    06-Jan-2014    100.01     94.01     88.01
    07-Jan-2014    100.02    94.02     88.02
    08-Jan-2014    100.02    94.02     88.02
    09-Jan-2014    100.04    94.04     88.04
    10-Jan-2014    100.06    94.06     88.05
    11-Jan-2014    100.08    94.08     88.07
    12-Jan-2014    100.11     94.1      88.1
    13-Jan-2014    100.11     94.1      88.1
    14-Jan-2014    100.12    94.11     88.11
    15-Jan-2014    100.12    94.11     88.11
```

Use `addBusinessCalendar` to indicate the holidays for the simulation period in the timetable TT.

```
TT = addBusinessCalendar(TT,Holidays=H);
```

Use `rollingreturns` with the `'Method'` name-value pair argument set to "`difference`" to compute the differences, that is, the period-over-period changes.

```
deltas = rollingreturns(TT,Period=calweeks(1),Method="difference");
head(deltas,15)
```

```
       Time        Price_Difference_1w    Price2_Difference_1w    Price3_Difference_1w
    ___________    ___________________    ____________________    ____________________

    01-Jan-2014            NaN                     NaN                     NaN        
    02-Jan-2014            NaN                     NaN                     NaN        
    03-Jan-2014            NaN                     NaN                     NaN        
    04-Jan-2014            NaN                     NaN                     NaN        
    05-Jan-2014            NaN                     NaN                     NaN        
    06-Jan-2014            NaN                     NaN                     NaN        
    07-Jan-2014            NaN                     NaN                     NaN        
    08-Jan-2014            NaN                     NaN                     NaN        
    09-Jan-2014           0.04                    0.04                    0.04        
    10-Jan-2014           0.06                    0.06                    0.05        
    11-Jan-2014           0.08                    0.08                    0.07        
    12-Jan-2014           0.11                     0.1                     0.1        
    13-Jan-2014            0.1                    0.09                    0.09        
    14-Jan-2014            0.1                    0.09                    0.09        
    15-Jan-2014            0.1                    0.09                    0.09        
```

Input Arguments

TT — Input timetable to update with business calendar awareness

timetable

Input timetable to update with business calendar awareness, specified as a `timetable`.

Data Types: `timetable`

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `TT = addBusinessCalendar(TT,Holidays=H)` replaces the default holidays stored in `Data_NYSE_Closures.mat` with the list of holidays `H`

`Holidays` — Alternate holidays and market closure dates
datetime vector

Alternate holidays and market closure dates, specified as a datetime vector. The dates in `Holidays` must be whole dates without `HH:MM:SS` components. No business is conducted on the dates in `Holidays`.

By default, `Holidays` is the New York Stock Exchange (NYSE) holidays and market closure dates. For more details, load the default holidays in `Data_NYSE_Closures.mat` and inspect the `NYSE` variable, or, if you have a Financial Toolbox license, see `holidays` and `isbusday`.

Tip If you have a Financial Toolbox license, you can generate alternate holiday schedules by using the `createholidays` function and performing this procedure:

1. Generate a new `holidays` function using `createholidays`.
2. Call the new `holidays` function to get the list of holidays.
3. Specify the alternate holidays to `addBusinessCalendar` by using the `Holidays` name-value argument.

Data Types: `datetime`

`Weekends` — Alternate weekend days on which no business is conducted
`[1 0 0 0 0 0 1]` or `["Sunday" "Saturday"]` (default) | logical vector | string vector

Alternate weekend days on which no business is conducted, specified as a length 7 logical vector or a string vector.

For a logical vector, `true` (`1`) entries indicate a weekend day and `false` (`0`) entries indicate a weekday, where entries correspond to Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, and Saturday.

Example: `Weekends=[1 0 0 0 0 1 1]` specifies that business is not conducted on Fridays through Sundays.

For a string vector, entries explicitly list the weekend days.

Example: `Weekends=["Friday" "Saturday" "Sunday"]`

Tip If business is conducted seven days per week, set `Weekends` to `[0 0 0 0 0 0 0]`.

Data Types: `logical`

Output Arguments

`TT` — Updated timetable `TT` with added business calendar awareness by a custom property
timetable

Updated timetable `TT` with added business calendar awareness by the custom property `BusinessCalendar`, returned as a `timetable`.

The custom property `BusinessCalendar` contains a data structure that contains a field `IsBusinessDay` that stores a callable function (`F`). The function `F` accepts a datetime matrix (`D`) and returns a logical indicator matrix (`I`) of the same size: `I = F(D)`. `true` (`1`) elements of `I` indicate that the corresponding element of `D` occurs on a business day; `false` (`0`) elements of `I` indicate otherwise.

Access the callable function `F` by using `F = TT.Properties.CustomProperties.BusinessCalendar.IsBusinessDay`.

Version History

Introduced in R2020b

See Also

`isbusday` | `rollingreturns` | `timetable`

rollingreturns

Period-over-period rolling returns or differences from prices

Syntax

```
returns = rollingreturns(TT)
returns = rollingreturns(___,Name,Value)
```

Description

`returns = rollingreturns(TT)` calculates period-over-period (PoP) returns or differences from corresponding prices. For each date in timetable `TT`, the return represents the PoP return of the corresponding price compared to the price one `Period` earlier.

`returns = rollingreturns(___,Name,Value)` specifies options using one or more optional name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Year-over-Year Returns from Daily Prices with Business Calendar Awareness

This example shows how to compute year-over-year rolling returns for five years of simulated daily prices and also includes business calendar awareness.

Simulate five years of daily prices and store the result in the timetable `TT`. Then, use `isbusday` to retain only data for New York Stock Exchange (NYSE) business dates.

```
rng(200,'twister')
time = (datetime(2014,1,1):caldays:datetime(2018,12,31))';
price = 100 + 0.1*(0:numel(time) - 1)'.*cumsum(randn(numel(time),1)/100);
price = round(price*100)/100;                  % Round prices to the nearest penny
TT = timetable(price,'RowTimes',time,'VariableNames',{'Prices'});
TT = TT(isbusday(TT.Properties.RowTimes),:);  % Retain only NYSE business days
head(TT,10)
```

```
       Time         Prices
    ___________     ______

    02-Jan-2014        100
    03-Jan-2014        100
    06-Jan-2014     100.01
    07-Jan-2014     100.02
    08-Jan-2014     100.02
    09-Jan-2014     100.04
    10-Jan-2014     100.06
    13-Jan-2014     100.11
    14-Jan-2014     100.12
    15-Jan-2014     100.12
```

Use `addBusinessCalendar` to add NYSE business calendar awareness. The business calendar logic determines if the date of the previous period is a business date, and if it is not, then the most recent business day preceding that date is found. For example, since 21-May-2016 is a Saturday and 22-May-2016 is a Sunday, year-over-year prices for Monday 22-May-2017 are compared to Friday 20-May-2016.

```
TT = addBusinessCalendar(TT);  % Add NYSE business calendar
```

Compute the year-over-year returns and display the last few prices and corresponding returns.

```
returns = rollingreturns(TT, 'Period', calyears);
tail([TT returns])
```

```
       Time        Prices    Prices_Return_1y
    ___________    ______    ________________

    19-Dec-2018    212.68        0.16941
    20-Dec-2018    215.54        0.19024
    21-Dec-2018    217.66        0.18648
    24-Dec-2018    221.42        0.20882
    26-Dec-2018    224.81        0.21473
    27-Dec-2018    222.17        0.19897
    28-Dec-2018    224.63        0.19142
    31-Dec-2018    224.37        0.19206
```

Year-over-Year Returns from Monthly Prices with End-of-Month Dates

Economic data is often reported on the last day of each month or quarter. So end-of-month ambiguities can arise when computing period-over-period returns for periods that exceed the periodicity at which data is reported.

Simulate five years of daily prices and store the result in the timetable TT.

```
rng(200,'twister')
time = (datetime(2014,1,1):caldays:datetime(2018,12,31))';
price = 100 + 0.1*(0:numel(time) - 1)'.*cumsum(randn(numel(time),1)/100);
price = round(price*100)/100;         % Round prices to the nearest penny
TT = timetable(price,'RowTimes',time,'VariableNames',{'Prices'});
head(TT,10)
```

```
       Time        Prices
    ___________    ______

    01-Jan-2014       100
    02-Jan-2014       100
    03-Jan-2014       100
    04-Jan-2014       100
    05-Jan-2014    100.01
    06-Jan-2014    100.01
    07-Jan-2014    100.02
    08-Jan-2014    100.02
    09-Jan-2014    100.04
    10-Jan-2014    100.06
```

Create a new timetable by sampling TT on the last day of each month to mimic monthly reporting.

```
monthEndDates = dateshift(TT.Time(1):calmonths:TT.Time(end),'end','month');
TT = TT(monthEndDates,:);   % Sample TT at end-of-month dates
head(TT,10)

       Time        Prices
    ___________    ______

    31-Jan-2014    100.47
    28-Feb-2014    100.93
    31-Mar-2014       102
    30-Apr-2014    102.28
    31-May-2014    103.22
    30-Jun-2014    103.92
    31-Jul-2014     102.2
    31-Aug-2014    104.79
    30-Sep-2014    103.11
    31-Oct-2014    105.29
```

Display a subset of the dates and compare a direct calculation of the dates in previous months to those shifted to the end of the month in which the previous period occurs.

```
dates = timerange(datetime(2016,2,29),datetime(2017,2,28),'month');
[TT.Time(dates) (TT.Time(dates) - calyears) dateshift(TT.Time(dates) - calyears,'end','month')]

ans = 13x3 datetime
   29-Feb-2016   28-Feb-2015   28-Feb-2015
   31-Mar-2016   31-Mar-2015   31-Mar-2015
   30-Apr-2016   30-Apr-2015   30-Apr-2015
   31-May-2016   31-May-2015   31-May-2015
   30-Jun-2016   30-Jun-2015   30-Jun-2015
   31-Jul-2016   31-Jul-2015   31-Jul-2015
   31-Aug-2016   31-Aug-2015   31-Aug-2015
   30-Sep-2016   30-Sep-2015   30-Sep-2015
   31-Oct-2016   31-Oct-2015   31-Oct-2015
   30-Nov-2016   30-Nov-2015   30-Nov-2015
   31-Dec-2016   31-Dec-2015   31-Dec-2015
   31-Jan-2017   31-Jan-2016   31-Jan-2016
   28-Feb-2017   28-Feb-2016   29-Feb-2016
```

Examine these results and notice that the dates in the second and third columns of the last row differ. Specifically, when the current date in the first column is `28-Feb-2017` the dates in the second and third columns differ because 2016 is a leap year. More generally, the dates differ whenever the month of the previous period has more days than the current month for which returns are computed. In this example, end-of-months dates present the following ambiguity. When the current date of interest is `28-Feb-2017`, should subtracting one calendar year produce `28-Feb-2016` or `29-Feb-2016`?

The correct answer depends on the application, and both approaches are valid use cases. This problem is exacerbated, for example, when working with end-of-monthly price data and computing month-over-month returns. To address the end-of-month ambiguity, the `rollingreturns` function supports an `EndOfMonth` flag.

```
returns = rollingreturns(TT, 'Period', calyears, 'EndOfMonth', true);
```

The `EndOfMonth` flag ensures that the `rollingreturns` function uses the correct end-of-month date of each calendar month. In this example, the return on `28-Feb-2017` is correctly computed from the price reported `29-Feb-2016` rather than `28-Feb-2016`.

```
[TT(dates,:) returns(dates,:)]
```

```
ans=13×2 timetable
       Time        Prices    Prices_Return_1y
    ___________    ______    ________________

    29-Feb-2016    135.59         0.21671    
    31-Mar-2016    138.47         0.25052    
    30-Apr-2016    131.44         0.11598    
    31-May-2016    129.34        0.083068    
    30-Jun-2016    133.86        0.077865    
    31-Jul-2016    132.78        0.046253    
    31-Aug-2016    140.32         0.11871    
    30-Sep-2016    136.52        0.087549    
    31-Oct-2016    141.27         0.10652    
    30-Nov-2016    140.76          0.1053    
    31-Dec-2016    135.96        0.057643    
    31-Jan-2017    129.52       0.0099025    
    28-Feb-2017    136.36       0.0056789    
```

Input Arguments

TT — Input timetable of prices

timetable

Input timetable of prices, specified as a `timetable`. The timetable `TT` must satisfy the following conditions:

- All observations in `TT` must be associated with whole dates specified as datetimes with no `HH:MM:SS` time component (no time-of-day component).
- `TT` dates must be sorted in ascending order.
- `TT` must have no duplicate dates.
- Each variable in `TT` must contain either a single numeric vector or a numeric matrix of prices. For example, suppose `TT` contains three variables of daily prices.

```
   Time         Price1    Price2          Prices
___________     ______    ______    ________________
24-Dec-2018     221.42    442.84    221.42    442.84
25-Dec-2018     220.62    441.24    220.62    441.24
26-Dec-2018     224.81    449.62    224.81    449.62
27-Dec-2018     222.17    444.34    222.17    444.34
28-Dec-2018     224.63    449.26    224.63    449.26
29-Dec-2018     225.36    450.72    225.36    450.72
30-Dec-2018     226.73    453.46    226.73    453.46
31-Dec-2018     224.37    448.74    224.37    448.74
```

The corresponding daily `returns` are formatted as three returns for the three price variables.

```
  Time         Price1_Return    Price2_Return       Prices_Return
___________    _____________    _____________    ________________________
24-Dec-2018          NaN              NaN             NaN         NaN
25-Dec-2018     -0.003613        -0.003613       -0.003613   -0.003613
26-Dec-2018      0.018992         0.018992        0.018992    0.018992
27-Dec-2018     -0.011743        -0.011743       -0.011743   -0.011743
```

```
    28-Dec-2018      0.011073         0.011073       0.011073    0.011073
    29-Dec-2018      0.003249         0.003249       0.003249    0.003249
    30-Dec-2018      0.006079         0.006079       0.006079    0.006079
    31-Dec-2018     -0.010409        -0.010409      -0.010409   -0.010409
```

Note To include business-calendar-awareness and account for nonbusiness days (for example, weekends, holidays, market closures), you must first use the `addBusinessCalendar` function to populate a custom property for the input `TT`. For example, to add business calendar logic to include only NYSE business days, you can use `TT = addBusinessCalendar(TT)`.

Data Types: `timetable`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `returns = rollingreturns(TT,'Period',calweeks(1),'EndOfMonth',true,'Method','continuous')`

`Period` — Period to compute period-over-period returns

`TT.Properties.TimeStep` (default) | scalar calendar duration

Period to compute period-over-period returns, specified as the comma-separated pair consisting of `'EndMonthRule'` and a scalar calendar duration (for example, `caldays`, `calweeks`, or `calmonths`).

The default is the time step defined in `TT` (`TT.Properties.TimeStep`), but only when `TT.Properties.TimeStep` is not `NaN`. If `TT.Properties.TimeStep` is `NaN`, then `Period` is required.

Data Types: `double`

`EndOfMonth` — End-of-month flag indicates whether prices at current date are compared to last date of month for previous `Period`

`false` (default) | value of `true` or `false`

End-of-month flag indicates whether the prices at the current date are compared to last date of month for the previous `Period`, specified as the comma-separated pair consisting of `'EndOfMonth'` and a scalar logical value of `true` or `false`.

- If you set `EndOfMonth` to `true` (logical `1`), meaning that the current prices are compared to end-of-month prices of the previous `Period`.
- If you set `EndOfMonth` to `false` (logical `0`), meaning that the current prices are compared to prices recorded on the actual date of the previous `Period`.

Note The `EndOfMonth` flag is intended to address end-of-month date calculations when computing the dates of a previous `Period` one or more months in the past.

For example, suppose you have monthly prices reported at the end of each month and want to compute year-over-year returns (that is, `Period = calyears(1)`). When the current date of interest is `28-Feb-2017`, setting `EndOfMonth = true` (logical 1) ensures that returns computed for `28-Feb-2017` compare the prices on `28-Feb-2017` to those on `29-Feb-2016` rather than `28-Feb-2016`.

Similarly, suppose you have monthly prices reported at the end of each month and want to compute month-over-month returns (that is, `Period = calmonths(1)`). When the current date of interest is `30-Apr-2020`, setting `EndOfMonth = true` (logical 1) ensures that returns computed for `30-Apr-2020` compare the prices on `30-Apr-2020` to those on `31-Mar-2020` rather than `30-Mar-2020`.

Data Types: `logical`

`Method` — Method for computing `returns` from prices

`'simple'` (default) | character vector with value `'simple'`, `'continuous'`, or `'difference'`

Method for computing `returns` from prices, specified as the comma-separated pair consisting of `'Method'` and a scalar character vector.

- `'simple'` — Compute simple (proportional) returns: `R(t) = P(t)/P(t-period) - 1`.
- `'continuous'` — Compute continuous (logarithmic) returns: `R(t) = log(P(t)/P(t-period))`.
- `'difference'` — Compute differences (period-over-period changes): `R(t) = P(t) - P(t-period)`

Data Types: `char`

Output Arguments

`returns` — Period-over-period decimal returns or differences

timetable

Period-over-period decimal returns or differences, returned as a timetable of the same size and format as the input argument `TT`. The returns or differences in row *t* are associated with the *t*th date in `TT` and represent the return or difference of the *t*th price *P*(*t*) relative to the price in the previous period *P*(*t*-period). If the date in the previous period is not found in `TT`, then the result is `NaN` to indicate a missing value.

Variable names in the output append `_Return` or `_Difference` to the variable names in `TT` for returns and differences, respectively, followed by the period used in the period-over-period results. For example, if `TT` has a variable named `ABC` and week-over-week returns are computed for a `Period` of `calweeks(1)`, the corresponding output variable is named `ABC_Returns_1w`.

`rollingreturns` is an aggregation function in which the frequency at which prices are recorded must equal or exceed that at which `returns` or differences are computed. For example, daily prices can be used to compute daily, weekly, or monthly `returns`, but computing daily `returns` from weekly or monthly prices generally makes no sense.

Algorithms

Period-over-period results are computed for every date in `TT` as follows:

1. For each date *t* in TT, the date *t*-period is computed.

   - If date *t*-period is a business date, then this date is the date "one period ago"
   - If date *t*-period is not a business date, then each calendar day preceding *t*-period is examined repeatedly until a business date is found, at which point this date is the date "one period ago," or the preceding date occurs prior to the first date in TT, at which point no previous business date exists in TT.
2. If the date "one period ago" is found in TT, then the corresponding price *P*(*t*-period) is recorded and the return *R*(*t*) is computed. However, if the date "one period ago" is not found in TT, then the previous price *P*(*t*-period) is assumed missing (that is, an implicit NaN), and the return *R*(*t*) = NaN.
3. The previous steps are repeated until the date *t*-period precedes the first date found in TT, at which point the algorithm terminates.

Version History

Introduced in R2020b

See Also

addBusinessCalendar | periodicreturns | timetable

simByMilstein

Simulate BM, GBM, CEV, HWV, SDEDDO, SDELD, SDEMRD sample paths by Milstein approximation

Syntax

```
[Paths,Times,Z] = simByMilstein(MDL,NPeriods)
[Paths,Times,Z] = simByMilstein( ___ ,Name=Value)
```

Description

[Paths,Times,Z] = simByMilstein(MDL,NPeriods) simulates NTrials sample paths of NVARS state variables driven by the BM, GBM, CEV, HWV, SDEDDO, SDELD, or SDEMRD process sources of risk over NPeriods consecutive observation periods, approximating continuous-time by the Milstein method.

simByMilstein provides a discrete-time approximation of the underlying generalized continuous-time process. The simulation is derived directly from the stochastic differential equation of motion; the discrete-time process approaches the true continuous-time process only in the limit as DeltaTimes approaches zero.

[Paths,Times,Z] = simByMilstein(___ ,Name=Value) specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

You can perform quasi-Monte Carlo simulations using the name-value arguments for MonteCarloMethod, QuasiSequence, and BrownianMotionMethod. For more information, see “Quasi-Monte Carlo Simulation” on page 14-62.

Examples

Quasi-Monte Carlo Simulation with Milstein Scheme Using GBM Model

This example shows how to use simByMilstein with a GBM model to perform a quasi-Monte Carlo simulation. Quasi-Monte Carlo simulation is a Monte Carlo simulation that uses quasi-random sequences instead of pseudo random numbers.

Create a univariategbm object to represent the model: $dX_t = 0.25X_tdt + 0.3X_tdW_t$.

```
GBM_obj = gbm(0.25, 0.3)  % (B = Return, Sigma)
```

```
GBM_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.

Perform a quasi-Monte Carlo simulation by using simByMilstein with the optional name-value arguments for MonteCarloMethod, QuasiSequence, and BrownianMotionMethod.

```
[paths,time,z] = simByMilstein(GBM_obj,10,ntrials=4096,montecarlomethod="quasi",quasisequence="so
```

Input Arguments

MDL — Stochastic differential equation model

BM object | GBM object | CEV object | HWV object | SDEDDO object | SDELD object | SDEMRD object

Stochastic differential equation model, specified as a BM, GBM, CEV, HWV, SDEDDO, SDELD, or SDEMRD object. You can use the following to create a MDL object:

- bm
- gbm
- cev
- hwv
- sdeddo
- sdeld
- sdemrd

Data Types: object

NPeriods — Number of simulation periods

positive scalar integer

Number of simulation periods, specified as a positive scalar integer. The value of NPeriods determines the number of rows of the simulated output series.

Data Types: double

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: [Paths,Times,Z] = simByMilstein(GBM_obj,NPeriods,DeltaTime=dt,Method="reflection")

Method — Method to handle negative values

'basic' (default) | character vector with values'basic', 'absorption', 'reflection', 'partial-truncation', 'full-truncation', or 'higham-mao' | string with values "basic", "absorption", "reflection", "partial-truncation", "full-truncation", or "higham-mao"

Method to handle negative values, specified as Method and a character vector or string with a supported value.

Note The `Method` argument is only supported when using a `CIR` object. For more information on creating a `CIR` object, see `cir`.

Data Types: `char` | `string`

`NTrials` — Simulated trials (sample paths) of `NPeriods` observations each

`1` (single path of correlated state variables) (default) | positive scalar integer

Simulated trials (sample paths) of `NPeriods` observations each, specified as `NTrials` and a positive scalar integer.

Data Types: `double`

`DeltaTimes` — Positive time increments between observations

`1` (default) | scalar | column vector

Positive time increments between observations, specified as the comma-separated pair consisting of `'DeltaTimes'` and a scalar or a `NPeriods`-by-`1` column vector.

`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.

Data Types: `double`

`NSteps` — Number of intermediate time steps within each time increment *dt* (specified as `DeltaTime`)

`1` (indicating no intermediate evaluation) (default) | positive scalar integer

Number of intermediate time steps within each time increment *dt* (specified as `DeltaTime`), specified as `NSteps` and a positive scalar integer.

The `simByMilestein` function 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 `simByMilstein` 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.

Data Types: `double`

`Antithetic` — Flag to indicate whether `simByMilstein` uses antithetic sampling to generate the Gaussian random variates

`False` (no antithetic sampling) (default) | logical with values `True` or `False`

Flag indicates whether `simByMilstein` uses antithetic sampling to generate the Gaussian random variates that drive the Brownian motion vector (Wiener processes). This argument is specified as `Antithetic` and a scalar logical flag with a value of `True` or `False`.

When you specify `True`, `simByMilstein` 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.

Note If you specify an input noise process (see `Z`), `simByMilstein` ignores the value of `Antithetic`.

Data Types: `logical`

Z — Direct specification of the dependent random noise process used to generate the Brownian motion vector
generates correlated Gaussian variates based on the `Correlation` member of the `SDE` object (default) | function | three-dimensional array of dependent random variates

Direct specification of the dependent random noise process used to generate the Brownian motion vector (Wiener process) that drives the simulation. This argument is specified as `Z` and a function or as an `(NPeriods □ NSteps)`-by-`NBrowns`-by-`NTrials` three-dimensional array of dependent random variates.

Note 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` | `function`

`StorePaths` — Flag that indicates how the output array `Paths` is stored and returned
`True` (default) | logical with values `True` or `False`

Flag that indicates how the output array `Paths` is stored and returned, specified as `StorePaths` and a scalar logical flag with a value of `True` or `False`.

If `StorePaths` is `True` (the default value) or is unspecified, `simByMilstein` returns `Paths` as a three-dimensional time series array.

If `StorePaths` is `False` (logical `0`), `simByMilstein` returns the `Paths` output array as an empty matrix.

Data Types: `logical`

`MonteCarloMethod` — Monte Carlo method to simulate stochastic processes
`"standard"` (default) | string with values `"standard"`, `"quasi"`, or `"randomized-quasi"` | character vector with values `'standard'`, `'quasi'`, or `'randomized-quasi'`

Monte Carlo method to simulate stochastic processes, specified as `MonteCarloMethod` and a string or character vector with one of the following values:

- `"standard"` — Monte Carlo using pseudo random numbers.
- `"quasi"` — Quasi-Monte Carlo using low-discrepancy sequences.
- `"randomized-quasi"` — Randomized quasi-Monte Carlo.

Note If you specify an input noise process (see `Z`), `simByMilstein` ignores the value of `MonteCarloMethod`.

Data Types: `string` | `char`

`QuasiSequence` — Low discrepancy sequence to drive the stochastic processes
`"sobol"` (default) | string with value `"sobol"` | character vector with value `'sobol'`

Low discrepancy sequence to drive the stochastic processes, specified as `QuasiSequence` and a string or character vector with one of the following values:

- `"sobol"` — Quasi-random low-discrepancy sequences that use a base of two to form successively finer uniform partitions of the unit interval and then reorder the coordinates in each dimension

Note If `MonteCarloMethod` option is not specified or specified as`"standard"`, `QuasiSequence` is ignored.

If you specify an input noise process (see `Z`), `simByMilstein` ignores the value of `QuasiSequence`.

Data Types: `string` | `char`

`BrownianMotionMethod` — Brownian motion construction method
`"standard"` (default) | string with value `"brownian-bridge"` or `"principal-components"` | character vector with value `'brownian-bridge'` or `'principal-components'`

Brownian motion construction method, specified as `BrownianMotionMethod` and a string or character vector with one of the following values:

- `"standard"` — The Brownian motion path is found by taking the cumulative sum of the Gaussian variates.
- `"brownian-bridge"` — The last step of the Brownian motion path is calculated first, followed by any order between steps until all steps have been determined.
- `"principal-components"` — The Brownian motion path is calculated by minimizing the approximation error.

Note If an input noise process is specified using the `Z` input argument, `BrownianMotionMethod` is ignored.

The starting point for a Monte Carlo simulation is the construction of a Brownian motion sample path (or Wiener path). Such paths are built from a set of independent Gaussian variates, using either standard discretization, Brownian-bridge construction, or principal components construction.

Both standard discretization and Brownian-bridge construction share the same variance and therefore the same resulting convergence when used with the `MonteCarloMethod` using pseudo random numbers. However, the performance differs between the two when the `MonteCarloMethod` option `"quasi"` is introduced, with faster convergence seen for `"brownian-bridge"` construction option and the fastest convergence when using the `"principal-components"` construction option.

Data Types: `string` | `char`

`Processes` — Sequence of end-of-period processes or state vector adjustments
`simByEuler` makes no adjustments and performs no processing (default) | function | cell array of functions

Sequence of end-of-period processes or state vector adjustments of the form, specified as `Processes` and a function or cell array of functions of the form

$$X_t = P(t, X_t)$$

The `simByMilstein` function 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, `simByMilstein` 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: `cell` | `function`

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`, `simByEuler` returns `Paths` as an empty matrix.

`Times` — Observation times associated with the simulated paths

column vector

Observation times associated with the simulated paths, returned as a `(NPeriods + 1)`-by-`1` column vector. Each element of `Times` is associated with the corresponding row of `Paths`.

`Z` — Dependent random variates used to generate the Brownian motion vector

array

Dependent random variates used to generate the Brownian motion vector (Wiener processes) that drive the simulation, returned as a `(NPeriods □ NSteps)`-by-`NBrowns`-by-`NTrials` three-dimensional time series array.

More About

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.

Algorithms

This function 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.

- 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).

Version History

Introduced in R2023a

References

[1] Milstein, G.N. "A Method of Second-Order Accuracy Integration of Stochastic Differential Equations."*Theory of Probability and Its Applications.* 23, 1978, pp. 396–401.

See Also

`simulate`

Topics

“Price American Basket Options Using Standard Monte Carlo and Quasi-Monte Carlo Simulation” on page 14-70
“SDEs” on page 14-2
“SDE Models” on page 14-7
“SDE Class Hierarchy” on page 14-5
“Quasi-Monte Carlo Simulation” on page 14-62
“Performance Considerations” on page 14-64

simByMilstein

Simulate CIR sample paths by Milstein approximation

Syntax

```
[Paths,Times,Z] = simByMilstein(MDL,NPeriods)
[Paths,Times,Z] = simByMilstein( ___ ,Name=Value)
```

Description

[Paths,Times,Z] = simByMilstein(MDL,NPeriods) simulates NTrials sample paths of NVARS state variables driven by the CIR process sources of risk over NPeriods consecutive observation periods, approximating continuous-time Cox-Ingersoll-Ross (CIR) by the Milstein method.

simByMilstein provides a discrete-time approximation of the underlying generalized continuous-time process. The simulation is derived directly from the stochastic differential equation of motion; the discrete-time process approaches the true continuous-time process only in the limit as DeltaTimes approaches zero.

[Paths,Times,Z] = simByMilstein(___ ,Name=Value) specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

You can perform quasi-Monte Carlo simulations using the name-value arguments for MonteCarloMethod, QuasiSequence, and BrownianMotionMethod. For more information, see “Quasi-Monte Carlo Simulation” on page 14-62.

Examples

Quasi-Monte Carlo Simulation with simByMilstein Scheme Using CIR Model

This example shows how to use simByMilstein with a CIR model to perform a quasi-Monte Carlo simulation. Quasi-Monte Carlo simulation is a Monte Carlo simulation that uses quasi-random sequences instead of pseudo random numbers.

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

```
cir_obj = cir(0.2, 0.1, 0.05)  % (Speed, Level, Sigma)
```

```
cir_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
```

Define the quasi-Monte Carlo simulation using the optional name-value arguments for `MonteCarloMethod`, `QuasiSequence`, and `BrownianMotionMethod`.

```
[paths,time,z] = simByMilstein(cir_obj,10,ntrials=4096,method="basic",montecarlomethod="quasi",q
```

Input Arguments

`MDL` — Stochastic differential equation model

CIR object

Stochastic differential equation model, specified as a `cir` object. You can create a `bates` object using `cir`.

Data Types: `object`

`NPeriods` — Number of simulation periods

positive scalar integer

Number of simulation periods, specified as a positive scalar integer. The value of `NPeriods` determines the number of rows of the simulated output series.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `[Paths,Times,Z] = simByMilstein(CIR_obj,NPeriods,DeltaTime=dt,Method="reflection")`

`Method` — Method to handle negative values

`'basic'` (default) | character vector with values`'basic'`, `'absorption'`, `'reflection'`, `'partialtruncation'`, `'fulltruncation'`, or `'higham-mao'` | string with values `"basic"`, `"absorption"`, `"reflection"`, `"partialtruncation"`, `"fulltruncation"`, or `"higham-mao"`

Method to handle negative values, specified as `Method` and a character vector or string with a supported value.

Note The `Method` argument is only supported when using a `CIR` object. For more information on creating a `CIR` object, see `cir`.

Data Types: char | string

NTrials — Simulated trials (sample paths) of NPeriods observations each
1 (single path of correlated state variables) (default) | positive scalar integer

Simulated trials (sample paths) of NPeriods observations each, specified as NTrials and a positive scalar integer.

Data Types: double

DeltaTimes — Positive time increments between observations
1 (default) | scalar | column vector

Positive time increments between observations, specified as the comma-separated pair consisting of 'DeltaTimes' and a scalar or a NPeriods-by-1 column vector.

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.

Data Types: double

NSteps — Number of intermediate time steps within each time increment *dt* (specified as DeltaTime)
1 (indicating no intermediate evaluation) (default) | positive scalar integer

Number of intermediate time steps within each time increment *dt* (specified as DeltaTime), specified as NSteps and a positive scalar integer.

The simByMilestein function 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 simByMilstein 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.

Data Types: double

Antithetic — Flag to indicate whether simByMilstein uses antithetic sampling to generate the Gaussian random variates
False (no antithetic sampling) (default) | logical with values True or False

Flag indicates whether simByMilstein uses antithetic sampling to generate the Gaussian random variates that drive the Brownian motion vector (Wiener processes). This argument is specified as Antithetic and a scalar logical flag with a value of True or False.

When you specify True, simByMilstein 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.

Note If you specify an input noise process (see Z), simByMilstein ignores the value of Antithetic.

Data Types: logical

Z — Direct specification of the dependent random noise process used to generate the Brownian motion vector
generates correlated Gaussian variates based on the `Correlation` member of the `SDE` object (default) | function | three-dimensional array of dependent random variates

Direct specification of the dependent random noise process used to generate the Brownian motion vector (Wiener process) that drives the simulation. This argument is specified as `Z` and a function or as an `(NPeriods ⨉ NSteps)`-by-`NBrowns`-by-`NTrials` three-dimensional array of dependent random variates.

Note 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` | `function`

`StorePaths` — Flag that indicates how the output array `Paths` is stored and returned
`True` (default) | logical with values `True` or `False`

Flag that indicates how the output array `Paths` is stored and returned, specified as `StorePaths` and a scalar logical flag with a value of `True` or `False`.

If `StorePaths` is `True` (the default value) or is unspecified, `simByMilstein` returns `Paths` as a three-dimensional time series array.

If `StorePaths` is `False` (logical `0`), `simByMilstein` returns the `Paths` output array as an empty matrix.

Data Types: `logical`

`MonteCarloMethod` — Monte Carlo method to simulate stochastic processes
`"standard"` (default) | string with values `"standard"`, `"quasi"`, or `"randomized-quasi"` | character vector with values `'standard'`, `'quasi'`, or `'randomized-quasi'`

Monte Carlo method to simulate stochastic processes, specified as `MonteCarloMethod` and a string or character vector with one of the following values:

- `"standard"` — Monte Carlo using pseudo random numbers.
- `"quasi"` — Quasi-Monte Carlo using low-discrepancy sequences.
- `"randomized-quasi"` — Randomized quasi-Monte Carlo.

Note If you specify an input noise process (see `Z`), `simByMilstein` ignores the value of `MonteCarloMethod`.

Data Types: `string` | `char`

`QuasiSequence` — Low discrepancy sequence to drive the stochastic processes
`"sobol"` (default) | string with value `"sobol"` | character vector with value `'sobol'`

Low discrepancy sequence to drive the stochastic processes, specified as `QuasiSequence` and a string or character vector with one of the following values:

- `"sobol"` — Quasi-random low-discrepancy sequences that use a base of two to form successively finer uniform partitions of the unit interval and then reorder the coordinates in each dimension

Note If `MonteCarloMethod` option is not specified or specified as`"standard"`, `QuasiSequence` is ignored.

If you specify an input noise process (see `Z`), `simByMilstein` ignores the value of `QuasiSequence`.

Data Types: `string` | `char`

`BrownianMotionMethod` — Brownian motion construction method
`"standard"` (default) | string with value `"brownian-bridge"` or `"principal-components"` | character vector with value `'brownian-bridge'` or `'principal-components'`

Brownian motion construction method, specified as `BrownianMotionMethod` and a string or character vector with one of the following values:

- `"standard"` — The Brownian motion path is found by taking the cumulative sum of the Gaussian variates.
- `"brownian-bridge"` — The last step of the Brownian motion path is calculated first, followed by any order between steps until all steps have been determined.
- `"principal-components"` — The Brownian motion path is calculated by minimizing the approximation error.

Note If an input noise process is specified using the `Z` input argument, `BrownianMotionMethod` is ignored.

The starting point for a Monte Carlo simulation is the construction of a Brownian motion sample path (or Wiener path). Such paths are built from a set of independent Gaussian variates, using either standard discretization, Brownian-bridge construction, or principal components construction.

Both standard discretization and Brownian-bridge construction share the same variance and therefore the same resulting convergence when used with the `MonteCarloMethod` using pseudo random numbers. However, the performance differs between the two when the `MonteCarloMethod` option `"quasi"` is introduced, with faster convergence seen for `"brownian-bridge"` construction option and the fastest convergence when using the `"principal-components"` construction option.

Data Types: `string` | `char`

`Processes` — Sequence of end-of-period processes or state vector adjustments
`simByEuler` makes no adjustments and performs no processing (default) | function | cell array of functions

Sequence of end-of-period processes or state vector adjustments of the form, specified as `Processes` and a function or cell array of functions of the form

$$X_t = P(t, X_t)$$

The `simByMilstein` function 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, `simByMilstein` 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: `cell` | `function`

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`, `simByEuler` returns `Paths` as an empty matrix.

`Times` — Observation times associated with the simulated paths

column vector

Observation times associated with the simulated paths, returned as a `(NPeriods + 1)`-by-`1` column vector. Each element of `Times` is associated with the corresponding row of `Paths`.

`Z` — Dependent random variates used to generate the Brownian motion vector

array

Dependent random variates used to generate the Brownian motion vector (Wiener processes) that drive the simulation, returned as a `(NPeriods  NSteps)`-by-`NBrowns`-by-`NTrials` three-dimensional time series array.

More About

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.

Algorithms

This function 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.

- 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).

Version History

Introduced in R2023a

References

[1] Milstein, G.N. "A Method of Second-Order Accuracy Integration of Stochastic Differential Equations."*Theory of Probability and Its Applications*. 23, 1978, pp. 396–401.

See Also

`simulate` | `cir`

Topics

“Price American Basket Options Using Standard Monte Carlo and Quasi-Monte Carlo Simulation” on page 14-70
“SDEs” on page 14-2
“SDE Models” on page 14-7
“SDE Class Hierarchy” on page 14-5

“Quasi-Monte Carlo Simulation” on page 14-62
“Performance Considerations” on page 14-64

simByMilstein

Simulate Heston sample paths by Milstein approximation

Syntax

```
[Paths,Times,Z] = simByMilstein(MDL,NPeriods)
[Paths,Times,Z] = simByMilstein( ___ ,Name=Value)
```

Description

[Paths,Times,Z] = simByMilstein(MDL,NPeriods) simulates NTrials sample paths of Heston bivariate models driven by two NBrowns Brownian motion sources of risk approximating continuous-time stochastic processes by Milstein scheme.

simByMilstein provides a discrete-time approximation of the underlying generalized continuous-time process. The simulation is derived directly from the stochastic differential equation of motion; the discrete-time process approaches the true continuous-time process only in the limit as DeltaTimes approaches zero.

[Paths,Times,Z] = simByMilstein(___ ,Name=Value) specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

You can perform quasi-Monte Carlo simulations using the name-value arguments for MonteCarloMethod, QuasiSequence, and BrownianMotionMethod. For more information, see “Quasi-Monte Carlo Simulation” on page 14-62.

Examples

Quasi-Monte Carlo Simulation with Milstein Scheme Using Heston Model

This example shows how to use simByMilstein with a Heston model to perform a quasi-Monte Carlo simulation. Quasi-Monte Carlo simulation is a Monte Carlo simulation that uses quasi-random sequences instead pseudo random numbers.

Define the parameters for the heston object.

```
Return = 0.03;
Level = 0.05;
Speed = 1.0;
Volatility = 0.2;

AssetPrice = 80;
V0 = 0.04;
Rho = -0.7;
StartState = [AssetPrice;V0];
Correlation = [1 Rho;Rho 1];
```

Create a heston object.

```
Heston = heston(Return,Speed,Level,Volatility,startstate=StartState,correlation=Correlation)
Heston = 
   Class HESTON: Heston Bivariate Stochastic Volatility
   ----------------------------------------------------
     Dimensions: State = 2, Brownian = 2
   ----------------------------------------------------
      StartTime: 0
     StartState: 2x1 double array
    Correlation: 2x2 double array
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
         Return: 0.03
          Speed: 1
          Level: 0.05
     Volatility: 0.2
```

Perform a quasi-Monte Carlo simulation by using `simByMilstein` with the optional name-value arguments for `MonteCarloMethod`, `QuasiSequence`, and `BrownianMotionMethod`.

```
[paths,time] = simByMilstein(Heston,10,ntrials=4096,MonteCarloMethod="quasi",QuasiSequence="sobo
```

Input Arguments

`MDL` — Stochastic differential equation model

`Heston` object

Stochastic differential equation model, specified as a `heston` object. You can create a `heston` object using `heston`.

Data Types: `object`

`NPeriods` — Number of simulation periods

positive scalar integer

Number of simulation periods, specified as a positive scalar integer. The value of `NPeriods` determines the number of rows of the simulated output series.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `[Paths,Times,Z] = simByMilstein(Heston_obj,NPeriods,NTrials=10,DeltaTimes=dt)`

`NTrials` — Simulated trials (sample paths)

`1` (single path of correlated state variables) (default) | positive scalar integer

Simulated trials (sample paths) of `NPeriods` observations each, specified as `NTrials` and a positive scalar integer.

Data Types: `double`

`DeltaTimes` — Positive time increments between observations

`1` (default) | scalar | column vector

Positive time increments between observations, specified as `DeltaTimes` and a scalar or an `NPeriods`-by-`1` column vector.

`DeltaTimes` 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.

Data Types: `double`

`NSteps` — Number of intermediate time steps within each time increment *dt*

`1` (indicating no intermediate evaluation) (default) | positive scalar integer

Number of intermediate time steps within each time increment *dt* (specified as `DeltaTimes`), specified as `NSteps` and a positive scalar integer.

The `simByMilstein` function 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 `simByMilstein` 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.

Data Types: `double`

`Antithetic` — Flag to use antithetic sampling to generate the Gaussian random variates

`false` (no antithetic sampling) (default) | logical with values `true` or `false`

Flag to use antithetic sampling to generate the Gaussian random variates that drive the Brownian motion vector (Wiener processes), specified as `Antithetic` and a scalar numeric or logical `1` (`true`) or `0` (`false`).

When you specify `true`, `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.

Note If you specify an input noise process (see `Z`), `simByMilstein` ignores the value of `Antithetic`.

Data Types: `logical`

`Z` — Direct specification of the dependent random noise process for generating Brownian motion vector

generates correlated Gaussian variates based on the `Correlation` member of the `SDE` object (default) | function | three-dimensional array of dependent random variates

Direct specification of the dependent random noise process for generating the Brownian motion vector (Wiener process) that drives the simulation, specified as `Z` and a function or as an `(NPeriods □ NSteps)`-by-`NBrowns`-by-`NTrials` three-dimensional array of dependent random variates.

Note 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 | function

StorePaths — Flag that indicates how Paths is stored and returned
true (default) | logical with values true or false

Flag that indicates how the output array Paths is stored and returned, specified as StorePaths and a scalar numeric or logical 1 (true) or 0 (false).

If StorePaths is true (the default value) or is unspecified, simByMilstein returns Paths as a three-dimensional time-series array.

If StorePaths is false (logical 0), simByMilstein returns Paths as an empty matrix.

Data Types: logical

MonteCarloMethod — Monte Carlo method to simulate stochastic processes
"standard" (default) | string with values "standard", "quasi", or "randomized-quasi" | character vector with values 'standard', 'quasi', or 'randomized-quasi'

Monte Carlo method to simulate stochastic processes, specified as MonteCarloMethod and a string or character vector with one of the following values:

- "standard" — Monte Carlo using pseudo random numbers.
- "quasi" — Quasi-Monte Carlo using low-discrepancy sequences.
- "randomized-quasi" — Randomized quasi-Monte Carlo.

Note If you specify an input noise process (see Z), simByMilstein ignores the value of MonteCarloMethod.

Data Types: string | char

QuasiSequence — Low discrepancy sequence to drive the stochastic processes
"sobol" (default) | string with value "sobol" | character vector with value 'sobol'

Low discrepancy sequence to drive the stochastic processes, specified as QuasiSequence and a string or character vector with one of the following values:

- "sobol" — Quasi-random low-discrepancy sequences that use a base of two to form successively finer uniform partitions of the unit interval and then reorder the coordinates in each dimension

Note If MonteCarloMethod option is not specified or specified as"standard", QuasiSequence is ignored.

If you specify an input noise process (see Z), simByMilstein ignores the value of QuasiSequence.

Data Types: `string` | `char`

`BrownianMotionMethod` — Brownian motion construction method

`"standard"` (default) | string with value `"brownian-bridge"` or `"principal-components"` | character vector with value `'brownian-bridge'` or `'principal-components'`

Brownian motion construction method, specified as `BrownianMotionMethod` and a string or character vector with one of the following values:

- `"standard"` — The Brownian motion path is found by taking the cumulative sum of the Gaussian variates.
- `"brownian-bridge"` — The last step of the Brownian motion path is calculated first, followed by any order between steps until all steps have been determined.
- `"principal-components"` — The Brownian motion path is calculated by minimizing the approximation error.

Note If an input noise process is specified using the `Z` input argument, `BrownianMotionMethod` is ignored.

The starting point for a Monte Carlo simulation is the construction of a Brownian motion sample path (or Wiener path). Such paths are built from a set of independent Gaussian variates, using either standard discretization, Brownian-bridge construction, or principal components construction.

Both standard discretization and Brownian-bridge construction share the same variance and therefore the same resulting convergence when used with the `MonteCarloMethod` using pseudo random numbers. However, the performance differs between the two when the `MonteCarloMethod` option `"quasi"` is introduced, with faster convergence seen for `"brownian-bridge"` construction option and the fastest convergence when using the `"principal-components"` construction option.

Data Types: `string` | `char`

`Processes` — Sequence of end-of-period processes or state vector adjustments

`simByMilstein` makes no adjustments and performs no processing (default) | function | cell array of functions

Sequence of end-of-period processes or state vector adjustments, specified as `Processes` and a function or cell array of functions of the form

$$X_t = P(t, X_t)$$

The `simByMilstein` function runs processing functions at each interpolation time. The functions must accept the current interpolation time t, and the current state vector X_t and return a state vector that can be an adjustment to the input state.

If you specify more than one processing function, `simByMilstein` 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.

The end-of-period `Processes` argument allows you to terminate a given trial early. At the end of each time step, `simByMilstein` 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 you to define a `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).

Data Types: `cell` | `function`

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 `StorePaths` is set to `false`, `simByMilstein` returns `Paths` as an empty matrix.

`Times` — Observation times associated with simulated paths

column vector

Observation times associated with the simulated paths, returned as an `(NPeriods + 1)`-by-`1` column vector. Each element of `Times` is associated with the corresponding row of `Paths`.

`Z` — Dependent random variates for generating Brownian motion vector

array

Dependent random variates for generating the Brownian motion vector (Wiener processes) that drive the simulation, returned as an `(NPeriods ⨉ NSteps)`-by-`NBrowns`-by-`NTrials` three-dimensional time-series array.

More About

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 that has the same expected value but a 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 other pairs, 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.

Algorithms

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.

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 `DeltaTimes` approaches zero.

Version History

Introduced in R2023a

References

[1] Milstein, G.N. "A Method of Second-Order Accuracy Integration of Stochastic Differential Equations."*Theory of Probability and Its Applications*. 23, 1978, pp. 396–401.

See Also

`heston` | `simulate`

Topics

“Price American Basket Options Using Standard Monte Carlo and Quasi-Monte Carlo Simulation” on page 14-70
“SDEs” on page 14-2
“SDE Models” on page 14-7
“SDE Class Hierarchy” on page 14-5
“Quasi-Monte Carlo Simulation” on page 14-62
“Performance Considerations” on page 14-64

simByMilstein

Simulate `Bates` sample paths by Milstein approximation

Syntax

```
[Paths,Times,Z,N] = simByMilstein(MDL,NPeriods)
[Paths,Times,Z,N] = simByMilstein( ___ ,Name=Value)
```

Description

`[Paths,Times,Z,N] = simByMilstein(MDL,NPeriods)` simulates `NTrials` sample paths of Bates or Heston bivariate models driven by `NBrowns` Brownian motion sources of risk and `NJUMPS` compound Poisson processes representing the arrivals of important events over `NPeriods` consecutive observation periods, approximating continuous-time stochastic processes by Milstein scheme.

`simByMilstein` provides a discrete-time approximation of the underlying generalized continuous-time process. The simulation is derived directly from the stochastic differential equation of motion; the discrete-time process approaches the true continuous-time process only in the limit as `DeltaTimes` approaches zero.

`[Paths,Times,Z,N] = simByMilstein( ___ ,Name=Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

You can perform quasi-Monte Carlo simulations using the name-value arguments for `MonteCarloMethod`, `QuasiSequence`, and `BrownianMotionMethod`. For more information, see “Quasi-Monte Carlo Simulation” on page 14-62.

Examples

Quasi-Monte Carlo Simulation with Milstein Scheme Using Bates Model

This example shows how to use `simByMilstein` with a Bates model to perform a quasi-Monte Carlo simulation. Quasi-Monte Carlo simulation is a Monte Carlo simulation that uses quasi-random sequences instead of pseudo random numbers.

Define the parameters for the `bates` object.

```
AssetPrice = 80;
Return = 0.03;
JumpMean = 0.02;
JumpVol = 0.08;
JumpFreq = 0.1;

V0 = 0.04;
Level = 0.05;
Speed = 1.0;
Volatility = 0.2;
Rho = -0.7;
```

```
StartState = [AssetPrice;V0];
Correlation = [1 Rho;Rho 1];
```

Create a `bates` object.

```
Bates = bates(Return, Speed, Level, Volatility, ...
                JumpFreq, JumpMean, JumpVol,startstate=StartState,correlation=Correlation)
```

```
Bates =
   Class BATES: Bates Bivariate Stochastic Volatility
   --------------------------------------------------
     Dimensions: State = 2, Brownian = 2
   --------------------------------------------------
      StartTime: 0
     StartState: 2x1 double array
    Correlation: 2x2 double array
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
         Return: 0.03
          Speed: 1
          Level: 0.05
     Volatility: 0.2
       JumpFreq: 0.1
       JumpMean: 0.02
        JumpVol: 0.08
```

Perform a quasi-Monte Carlo simulation by using `simByMilstein` with the optional name-value arguments for `MonteCarloMethod`, `QuasiSequence`, and `BrownianMotionMethod`.

```
[paths,time,z,n] = simByMilstein(Bates,10,ntrials=4096,montecarlomethod="quasi",quasisequence="so
```

Input Arguments

`MDL` — Stochastic differential equation model

`Bates` object

Stochastic differential equation model, specified as a `bates` object. You can create a `bates` object using `bates`.

Data Types: `object`

`NPeriods` — Number of simulation periods

positive scalar integer

Number of simulation periods, specified as a positive scalar integer. The value of `NPeriods` determines the number of rows of the simulated output series.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `[Paths,Times,Z,N] = simByMilstein(Bates_obj,NPeriods,NTrials=10,DeltaTimes=dt)`

`NTrials` — Simulated trials (sample paths)

`1` (single path of correlated state variables) (default) | positive scalar integer

Simulated trials (sample paths) of `NPeriods` observations each, specified as `NTrials` and a positive scalar integer.

Data Types: `double`

`DeltaTimes` — Positive time increments between observations

`1` (default) | scalar | column vector

Positive time increments between observations, specified as `DeltaTimes` and a scalar or an `NPeriods`-by-`1` column vector.

`DeltaTimes` 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.

Data Types: `double`

`NSteps` — Number of intermediate time steps within each time increment *dt*

`1` (indicating no intermediate evaluation) (default) | positive scalar integer

Number of intermediate time steps within each time increment *dt* (specified as `DeltaTimes`), specified as `NSteps` and a positive scalar integer.

The `simByMilstein` function 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 `simByMilstein` 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.

Data Types: `double`

`Antithetic` — Flag to use antithetic sampling to generate the Gaussian random variates

`false` (no antithetic sampling) (default) | logical with values `true` or `false`

Flag to use antithetic sampling to generate the Gaussian random variates that drive the Brownian motion vector (Wiener processes), specified as `Antithetic` and a scalar numeric or logical `1` (`true`) or `0` (`false`).

When you specify `true`, `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.

Note If you specify an input noise process (see `Z`), `simByMilstein` ignores the value of `Antithetic`.

Data Types: `logical`

Z — Direct specification of the dependent random noise process for generating Brownian motion vector

generates correlated Gaussian variates based on the `Correlation` member of the `SDE` object (default) | function | three-dimensional array of dependent random variates

Direct specification of the dependent random noise process for generating the Brownian motion vector (Wiener process) that drives the simulation, specified as `Z` and a function or as an `(NPeriods  NSteps)`-by-`NBrowns`-by-`NTrials` three-dimensional array of dependent random variates.

Note 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` | `function`

N — Dependent random counting process for generating number of jumps

random numbers from Poisson distribution with parameter `JumpFreq` from a `bates` object (default) | three-dimensional array | function

Dependent random counting process for generating the number of jumps, specified as `N` and a function or an (`NPeriods` `NSteps`) -by-`NJumps`-by-`NTrials` three-dimensional array of dependent random variates.

Note If you specify a function, `N` must return an `NJumps`-by-`1` column vector, and you must call it with two inputs: a real-valued scalar observation time t followed by an `NVars`-by-`1` state vector X_t.

Data Types: `double` | `function`

`StorePaths` — Flag that indicates how `Paths` is stored and returned

`true` (default) | logical with values `true` or `false`

Flag that indicates how the output array `Paths` is stored and returned, specified as `StorePaths` and a scalar numeric or logical `1` (`true`) or `0` (`false`).

If `StorePaths` is `true` (the default value) or is unspecified, `simByMilstein` returns `Paths` as a three-dimensional time-series array.

If `StorePaths` is `false` (logical `0`), `simByMilstein` returns `Paths` as an empty matrix.

Data Types: `logical`

`MonteCarloMethod` — Monte Carlo method to simulate stochastic processes

`"standard"` (default) | string with values `"standard"`, `"quasi"`, or `"randomized-quasi"` | character vector with values `'standard'`, `'quasi'`, or `'randomized-quasi'`

Monte Carlo method to simulate stochastic processes, specified as `MonteCarloMethod` and a string or character vector with one of the following values:

- "standard" — Monte Carlo using pseudo random numbers.
- "quasi" — Quasi-Monte Carlo using low-discrepancy sequences.
- "randomized-quasi" — Randomized quasi-Monte Carlo.

Note If you specify an input noise process (see Z), simByMilstein ignores the value of MonteCarloMethod.

Data Types: string | char

QuasiSequence — Low discrepancy sequence to drive the stochastic processes
"sobol" (default) | string with value "sobol" | character vector with value 'sobol'

Low discrepancy sequence to drive the stochastic processes, specified as QuasiSequence and a string or character vector with one of the following values:

- "sobol" — Quasi-random low-discrepancy sequences that use a base of two to form successively finer uniform partitions of the unit interval and then reorder the coordinates in each dimension

Note If MonteCarloMethod option is not specified or specified as"standard", QuasiSequence is ignored.

If you specify an input noise process (see Z), simByMilstein ignores the value of QuasiSequence.

Data Types: string | char

BrownianMotionMethod — Brownian motion construction method
"standard" (default) | string with value "brownian-bridge" or "principal-components" | character vector with value 'brownian-bridge' or 'principal-components'

Brownian motion construction method, specified as BrownianMotionMethod and a string or character vector with one of the following values:

- "standard" — The Brownian motion path is found by taking the cumulative sum of the Gaussian variates.
- "brownian-bridge" — The last step of the Brownian motion path is calculated first, followed by any order between steps until all steps have been determined.
- "principal-components" — The Brownian motion path is calculated by minimizing the approximation error.

Note If an input noise process is specified using the Z input argument, BrownianMotionMethod is ignored.

The starting point for a Monte Carlo simulation is the construction of a Brownian motion sample path (or Wiener path). Such paths are built from a set of independent Gaussian variates, using either standard discretization, Brownian-bridge construction, or principal components construction.

Both standard discretization and Brownian-bridge construction share the same variance and therefore the same resulting convergence when used with the MonteCarloMethod using pseudo random numbers. However, the performance differs between the two when the `MonteCarloMethod` option `"quasi"` is introduced, with faster convergence seen for `"brownian-bridge"` construction option and the fastest convergence when using the `"principal-components"` construction option.

Data Types: `string` | `char`

`Processes` — Sequence of end-of-period processes or state vector adjustments
`simByMilstein` makes no adjustments and performs no processing (default) | function | cell array of functions

Sequence of end-of-period processes or state vector adjustments, specified as `Processes` and a function or cell array of functions of the form

$$X_t = P(t, X_t)$$

The `simByMilstein` function runs processing functions at each interpolation time. The functions must accept the current interpolation time t, and the current state vector X_t and return a state vector that can be an adjustment to the input state.

If you specify more than one processing function, `simByMilstein` 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.

The end-of-period `Processes` argument allows you to terminate a given trial early. At the end of each time step, `simByMilstein` 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 you to define a `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).

Data Types: `cell` | `function`

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 `StorePaths` is set to `false`, `simByMilstein` returns `Paths` as an empty matrix.

`Times` — Observation times associated with simulated paths
column vector

Observation times associated with the simulated paths, returned as an `(NPeriods + 1)`-by-`1` column vector. Each element of `Times` is associated with the corresponding row of `Paths`.

`Z` — Dependent random variates for generating Brownian motion vector
array

Dependent random variates for generating the Brownian motion vector (Wiener processes) that drive the simulation, returned as an `(NPeriods ⨉ NSteps)`-by-`NBrowns`-by-`NTrials` three-dimensional time-series array.

`N` — Dependent random variates for generating jump counting process vector

array

Dependent random variates used to generate the jump counting process vector, returned as an `(NPeriods  NSteps)`-by-`NJumps`-by-`NTrials` three-dimensional time series array.

More About

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 that has the same expected value but a 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 other pairs, 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.

Algorithms

Bates models are bivariate composite models. Each Bates model consists of two coupled univariate models.

- One model is a geometric Brownian motion (`gbm`) model with a stochastic volatility function and jumps.

  $$dX_{1t} = B(t)X_{1t}dt + \sqrt{X_{2t}}X_{1t}dW_{1t} + Y(t)X_{1t}dN_t$$

  This model usually corresponds to a price process whose volatility (variance rate) is governed by the second univariate model.
- The other model is 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 Bates price process.

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 `DeltaTimes` approaches zero.

Version History

Introduced in R2023a

References

[1] Milstein, G.N. "A Method of Second-Order Accuracy Integration of Stochastic Differential Equations."*Theory of Probability and Its Applications*. 23, 1978, pp. 396–401.

See Also

`bates` | `simulate`

Topics

"Price American Basket Options Using Standard Monte Carlo and Quasi-Monte Carlo Simulation" on page 14-70
"SDEs" on page 14-2
"SDE Models" on page 14-7
"SDE Class Hierarchy" on page 14-5
"Quasi-Monte Carlo Simulation" on page 14-62
"Performance Considerations" on page 14-64

simByMilstein

Simulate `Merton` sample paths by Milstein approximation

Syntax

```
[Paths,Times,Z,N] = simByMilstein(MDL,NPeriods)
[Paths,Times,Z,N] = simByMilstein(___,Name=Value)
```

Description

`[Paths,Times,Z,N] = simByMilstein(MDL,NPeriods)` simulates `NTrials` sample paths of `NVars` correlated state variables driven by `NBrowns` Brownian motion sources of risk and `NJumps` compound Poisson processes representing the arrivals of important events over `NPeriods` consecutive observation periods. The simulation approximates the continuous-time Merton jump diffusion process by the Euler approach.

`[Paths,Times,Z,N] = simByMilstein(___,Name=Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

You can perform quasi-Monte Carlo simulations using the name-value arguments for `MonteCarloMethod`, `QuasiSequence`, and `BrownianMotionMethod`. For more information, see “Quasi-Monte Carlo Simulation” on page 14-62.

Examples

Quasi-Monte Carlo Simulation with Milstein Scheme Using Merton Model

This example shows how to use `simByMilstein` with a Merton model to perform a quasi-Monte Carlo simulation. Quasi-Monte Carlo simulation is a Monte Carlo simulation that uses quasi-random sequences instead pseudo random numbers.

Define the parameters for the `merton` object.

```
AssetPrice = 80;
Return = 0.03;
Sigma = 0.16;
JumpMean = 0.02;
JumpVol = 0.08;
JumpFreq = 2;
```

Create a `merton` object.

```
mertonObj = merton(Return,Sigma,JumpFreq,JumpMean,JumpVol,StartState=AssetPrice)
```

```
mertonObj = 
   Class MERTON: Merton Jump Diffusion
   ----------------------------------------
     Dimensions: State = 1, Brownian = 1
   ----------------------------------------
      StartTime: 0
     StartState: 80
    Correlation: 1
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
          Sigma: 0.16
         Return: 0.03
       JumpFreq: 2
       JumpMean: 0.02
        JumpVol: 0.08
```

Perform a quasi-Monte Carlo simulation by using `simByMilstein` with the optional name-value arguments for `MonteCarloMethod`, `QuasiSequence`, and `BrownianMotionMethod`.

```
[paths,times] = simByMilstein(mertonObj,10,Ntrials=4096,MonteCarloMethod="quasi",QuasiSequence="
```

Input Arguments

`MDL` — Stochastic differential equation model

`Merton` object

Stochastic differential equation model, specified as a `merton` object. You can create a `merton` object using `merton`.

Data Types: `object`

`NPeriods` — Number of simulation periods

positive scalar integer

Number of simulation periods, specified as a positive scalar integer. The value of `NPeriods` determines the number of rows of the simulated output series.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `[Paths,Times,Z,N] = simByMilstein(Merton_obj,NPeriods,NTrials=1000,DeltaTimes=dt,NSteps=10)`

`NTrials` — Simulated trials (sample paths)

`1` (single path of correlated state variables) (default) | positive scalar integer

Simulated trials (sample paths) of `NPeriods` observations each, specified as the comma-separated pair consisting of `'NTrials'` and a positive scalar integer.

Data Types: `double`

`DeltaTimes` — Positive time increments between observations

`1` (default) | scalar | column vector

Positive time increments between observations, specified as the comma-separated pair consisting of `'DeltaTimes'` and a scalar or a `NPeriods`-by-`1` column vector.

`DeltaTimes` 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.

Data Types: `double`

`NSteps` — Number of intermediate time steps within each time increment
`1` (indicating no intermediate evaluation) (default) | positive scalar integer

Number of intermediate time steps within each time increment *dt* (specified as `DeltaTimes`), specified as the comma-separated pair consisting of `'NSteps'` and a positive scalar integer.

The `simByEuler` function 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.

Data Types: `double`

`Antithetic` — Flag to use antithetic sampling to generate the Gaussian random variates
`false` (no antithetic sampling) (default) | logical with values `true` or `false`

Flag to use antithetic sampling to generate the Gaussian random variates that drive the Brownian motion vector (Wiener processes), specified as the comma-separated pair consisting of `'Antithetic'` and a scalar numeric or logical `1` (`true`) or `0` (`false`).

When you specify `true`, `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.

Note If you specify an input noise process (see `Z`), `simByEuler` ignores the value of `Antithetic`.

Data Types: `logical`

`Z` — Direct specification of the dependent random noise process for generating Brownian motion vector
generates correlated Gaussian variates based on the `Correlation` member of the `SDE` object (default) | function | three-dimensional array of dependent random variates

Direct specification of the dependent random noise process for generating the Brownian motion vector (Wiener process) that drives the simulation, specified as the comma-separated pair consisting of `'Z'` and a function or as an `(NPeriods ∗ NSteps)`-by-`NBrowns`-by-`NTrials` three-dimensional array of dependent random variates.

Note 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` | `function`

`N` — Dependent random counting process for generating number of jumps
random numbers from Poisson distribution with parameter `JumpFreq` from `merton` object (default) | three-dimensional array | function

Dependent random counting process for generating the number of jumps, specified as the comma-separated pair consisting of `'N'` and a function or an (`NPeriods` □ `NSteps`) -by-`NJumps`-by-`NTrials` three-dimensional array of dependent random variates.

If you specify a function, `N` must return an `NJumps`-by-`1` column vector, and you must call it with two inputs: a real-valued scalar observation time *t* followed by an `NVars`-by-`1` state vector X_t.

Data Types: `double` | `function`

`StorePaths` — Flag that indicates how `Paths` is stored and returned
`true` (default) | logical with values `true` or `false`

Flag that indicates how the output array `Paths` is stored and returned, specified as the comma-separated pair consisting of `'StorePaths'` and a scalar numeric or logical `1` (`true`) or `0` (`false`).

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 `Paths` as an empty matrix.

Data Types: `logical`

`MonteCarloMethod` — Monte Carlo method to simulate stochastic processes
`"standard"` (default) | string with values `"standard"`, `"quasi"`, or `"randomized-quasi"` | character vector with values `'standard'`, `'quasi'`, or `'randomized-quasi'`

Monte Carlo method to simulate stochastic processes, specified as the comma-separated pair consisting of `'MonteCarloMethod'` and a string or character vector with one of the following values:

- `"standard"` — Monte Carlo using pseudo random numbers.
- `"quasi"` — Quasi-Monte Carlo using low-discrepancy sequences.
- `"randomized-quasi"` — Randomized quasi-Monte Carlo.

Note If you specify an input noise process (see `Z` and `N`), `simByEuler` ignores the value of `MonteCarloMethod`.

Data Types: `string` | `char`

`QuasiSequence` — Low discrepancy sequence to drive the stochastic processes
`"sobol"` (default) | string with value `"sobol"` | character vector with value `'sobol'`

Low discrepancy sequence to drive the stochastic processes, specified as the comma-separated pair consisting of `'QuasiSequence'` and a string or character vector with one of the following values:

- `"sobol"` — Quasi-random low-discrepancy sequences that use a base of two to form successively finer uniform partitions of the unit interval and then reorder the coordinates in each dimension

Note If MonteCarloMethod option is not specified or specified as"standard", QuasiSequence is ignored.

Data Types: string | char

BrownianMotionMethod — Brownian motion construction method
"standard" (default) | string with value "brownian-bridge" or "principal-components" | character vector with value 'brownian-bridge' or 'principal-components'

Brownian motion construction method, specified as the comma-separated pair consisting of 'BrownianMotionMethod' and a string or character vector with one of the following values:

- "standard" — The Brownian motion path is found by taking the cumulative sum of the Gaussian variates.
- "brownian-bridge" — The last step of the Brownian motion path is calculated first, followed by any order between steps until all steps have been determined.
- "principal-components" — The Brownian motion path is calculated by minimizing the approximation error.

Note If an input noise process is specified using the Z input argument, BrownianMotionMethod is ignored.

The starting point for a Monte Carlo simulation is the construction of a Brownian motion sample path (or Wiener path). Such paths are built from a set of independent Gaussian variates, using either standard discretization, Brownian-bridge construction, or principal components construction.

Both standard discretization and Brownian-bridge construction share the same variance and therefore the same resulting convergence when used with the MonteCarloMethod using pseudo random numbers. However, the performance differs between the two when the MonteCarloMethod option "quasi" is introduced, with faster convergence seen for "brownian-bridge" construction option and the fastest convergence when using the "principal-components" construction option.

Data Types: string | char

Processes — Sequence of end-of-period processes or state vector adjustments
simByEuler makes no adjustments and performs no processing (default) | function | cell array of functions

Sequence of end-of-period processes or state vector adjustments, specified as the comma-separated pair consisting of 'Processes' and a function or cell array of functions of the form

$$X_t = P(t, X_t)$$

The simByEuler function runs processing functions at each interpolation time. The functions must accept the current interpolation time t, and the current state vector X_t and return a state vector that can 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.

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 you to define a `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).

Data Types: `cell` | `function`

Output Arguments

`Paths` — Simulated paths of correlated state variables

array

Simulated paths of correlated state variables, returned as an `(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 `StorePaths` is set to `false`, `simByEuler` returns `Paths` as an empty matrix.

`Times` — Observation times associated with simulated paths

column vector

Observation times associated with the simulated paths, returned as an `(NPeriods + 1)`-by-`1` column vector. Each element of `Times` is associated with the corresponding row of `Paths`.

`Z` — Dependent random variates for generating Brownian motion vector

array

Dependent random variates used to generate the Brownian motion vector (Wiener processes) that drive the simulation, returned as an `(NPeriods ⨉ NSteps)`-by-`NBrowns`-by-`NTrials` three-dimensional time-series array.

`N` — Dependent random variates used for generating jump counting process vector

array

Dependent random variates for generating the jump counting process vector, returned as an `(NPeriods ⨉ NSteps)`-by-`NJumps`-by-`NTrials` three-dimensional time-series array.

More About

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 that has the same expected value but a 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 other pairs, 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.

Algorithms

This function simulates any vector-valued SDE of the following form:

$$dX_t = B(t, X_t)X_t dt + D(t, X_t)V(t, x_t)dW_t + Y(t, X_t, N_t)X_t dN_t$$

Here:

- X_t is an `NVars`-by-`1` state vector of process variables.
- $B(t,X_t)$ is an `NVars`-by-`NVars` matrix of generalized expected instantaneous rates of return.
- $D(t,X_t)$ is an `NVars`-by-`NVars` diagonal matrix in which each element along the main diagonal is the corresponding element of the state vector.
- $V(t,X_t)$ is an `NVars`-by-`NVars` matrix of instantaneous volatility rates.
- dW_t is an `NBrowns`-by-`1` Brownian motion vector.
- $Y(t,X_t,N_t)$ is an `NVars`-by-`NJumps` matrix-valued jump size function.
- dN_t is an `NJumps`-by-`1` counting process vector.

`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.

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 `DeltaTimes` approaches zero.

Version History

Introduced in R2023a

References

[1] Milstein, G.N. "A Method of Second-Order Accuracy Integration of Stochastic Differential Equations."*Theory of Probability and Its Applications.* 23, 1978, pp. 396–401.

See Also

`merton` | `simulate`

Topics

"Price American Basket Options Using Standard Monte Carlo and Quasi-Monte Carlo Simulation" on page 14-70
"SDEs" on page 14-2
"SDE Models" on page 14-7
"SDE Class Hierarchy" on page 14-5
"Quasi-Monte Carlo Simulation" on page 14-62
"Performance Considerations" on page 14-64

simByTransition

Simulate `Bates` sample paths with transition density

Syntax

```
[Paths,Times] = simByTransition(MDL,NPeriods)
[Paths,Times] = simByTransition( ___ ,Name,Value)
```

Description

`[Paths,Times] = simByTransition(MDL,NPeriods)` simulates `NTrials` of Bates bivariate models driven by two Brownian motion sources of risk and one compound Poisson process representing the arrivals of important events over `NPeriods` consecutive observation periods. `simByTransition` approximates continuous-time stochastic processes by the transition density.

`[Paths,Times] = simByTransition( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

You can perform quasi-Monte Carlo simulations using the name-value arguments for `MonteCarloMethod`, `QuasiSequence`, and `BrownianMotionMethod`. For more information, see "Quasi-Monte Carlo Simulation" on page 14-62.

Examples

Use `simByTransition` with bates Object

Simulate Bates sample paths with transition density.

Define the parameters for the `bates` object.

```
AssetPrice = 80;
Return = 0.03;
JumpMean = 0.02;
JumpVol = 0.08;
JumpFreq = 0.1;
V0 = 0.04;
Level = 0.05;
Speed = 1.0;
Volatility = 0.2;
Rho = -0.7;
StartState = [AssetPrice;V0];
Correlation = [1 Rho;Rho 1];
```

Create a `bates` object.

```
batesObj = bates(Return, Speed, Level, Volatility,...
                JumpFreq, JumpMean, JumpVol,'startstate',StartState,...
                'correlation',Correlation)
```

```
batesObj = 
   Class BATES: Bates Bivariate Stochastic Volatility
```

```
  --------------------------------------------------
    Dimensions: State = 2, Brownian = 2
  --------------------------------------------------
     StartTime: 0
    StartState: 2x1 double array
   Correlation: 2x2 double array
         Drift: drift rate function F(t,X(t))
     Diffusion: diffusion rate function G(t,X(t))
    Simulation: simulation method/function simByEuler
        Return: 0.03
         Speed: 1
         Level: 0.05
    Volatility: 0.2
      JumpFreq: 0.1
      JumpMean: 0.02
       JumpVol: 0.08
```

Define the simulation parameters.

```
nPeriods = 5;   % Simulate sample paths over the next five years
Paths = simByTransition(batesObj,nPeriods);
Paths
```

```
Paths = 6×2

   80.0000    0.0400
   99.5724    0.0201
  124.2066    0.0176
   56.5051    0.1806
   80.4545    0.1732
   94.7887    0.1169
```

Quasi-Monte Carlo Simulation Using Bates Model

This example shows how to use `simByTransition` with a Bates model to perform a quasi-Monte Carlo simulation. Quasi-Monte Carlo simulation is a Monte Carlo simulation that uses quasi-random sequences instead pseudo random numbers.

Define the parameters for the `bates` object.

```
AssetPrice = 80;
Return = 0.03;
JumpMean = 0.02;
JumpVol = 0.08;
JumpFreq = 0.1;
V0 = 0.04;
Level = 0.05;
Speed = 1.0;
Volatility = 0.2;
Rho = -0.7;
StartState = [AssetPrice;V0];
Correlation = [1 Rho;Rho 1];
```

Create a `bates` object.

```
Bates = bates(Return, Speed, Level, Volatility,...
                JumpFreq, JumpMean, JumpVol,'startstate',StartState,...
                'correlation',Correlation)
```

```
Bates = 
   Class BATES: Bates Bivariate Stochastic Volatility
   --------------------------------------------------
     Dimensions: State = 2, Brownian = 2
   --------------------------------------------------
      StartTime: 0
     StartState: 2x1 double array
    Correlation: 2x2 double array
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
         Return: 0.03
          Speed: 1
          Level: 0.05
     Volatility: 0.2
       JumpFreq: 0.1
       JumpMean: 0.02
        JumpVol: 0.08
```

Perform a quasi-Monte Carlo simulation by using `simByTransition` with the optional name-value argument for `'MonteCarloMethod'`, `'QuasiSequence'`, and `'BrownianMotionMethod'`.

```
[paths,time] = simByTransition(Bates,10,'ntrials',4096,'MonteCarloMethod','quasi','QuasiSequence
```

Input Arguments

`MDL` — Stochastic differential equation model

`bates` object

Stochastic differential equation model, specified as a `bates` object. For more information on creating a `bates` object, see `bates`.

Data Types: `object`

`NPeriods` — Number of simulation periods

positive scalar integer

Number of simulation periods, specified as a positive scalar integer. The value of `NPeriods` determines the number of rows of the simulated output series.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Paths,Times] = simByTransition(Bates,NPeriods,'DeltaTimes',dt)`

`NTrials` — Simulated trials (sample paths)

`1` (single path of correlated state variables) (default) | positive integer

Simulated trials (sample paths) of `NPeriods` observations each, specified as the comma-separated pair consisting of `'NTrials'` and a positive scalar integer.

Data Types: `double`

`DeltaTimes` — Positive time increments between observations

`1` (default) | scalar | column vector

Positive time increments between observations, specified as the comma-separated pair consisting of `'DeltaTimes'` and a scalar or `NPeriods`-by-`1` column vector.

`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.

Data Types: `double`

`NSteps` — Number of intermediate time steps

`1` (indicating no intermediate evaluation) (default) | positive integer

Number of intermediate time steps within each time increment *dt* (defined as `DeltaTimes`), specified as the comma-separated pair consisting of `'NSteps'` and a positive scalar integer.

The `simByTransition` function 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 `simByTransition` does not report the output state vector at these intermediate points, the refinement improves accuracy by enabling the simulation to more closely approximate the underlying continuous-time process.

Data Types: `double`

`StorePaths` — Flag for storage and return method

`True` (default) | logical with values `True` or `False`

Flag for storage and return method that indicates how the output array `Paths` is stored and returned, specified as the comma-separated pair consisting of `'StorePaths'` and a scalar logical flag with a value of `True` or `False`.

- If `StorePaths` is `True` (the default value) or is unspecified, then `simByTransition` returns `Paths` as a three-dimensional time series array.
- If `StorePaths` is `False` (logical `0`), then `simByTransition` returns the `Paths` output array as an empty matrix.

Data Types: `logical`

`MonteCarloMethod` — Monte Carlo method to simulate stochastic processes

`"standard"` (default) | string with values `"standard"`, `"quasi"`, or `"randomized-quasi"` | character vector with values `'standard'`, `'quasi'`, or `'randomized-quasi'`

Monte Carlo method to simulate stochastic processes, specified as the comma-separated pair consisting of `'MonteCarloMethod'` and a string or character vector with one of the following values:

- `"standard"` — Monte Carlo using pseudo random numbers.
- `"quasi"` — Quasi-Monte Carlo using low-discrepancy sequences.
- `"randomized-quasi"` — Randomized quasi-Monte Carlo.

Data Types: `string` | `char`

`QuasiSequence` — Low discrepancy sequence to drive the stochastic processes
`"sobol"` (default) | string with value `"sobol"` | character vector with value `'sobol'`

Low discrepancy sequence to drive the stochastic processes, specified as the comma-separated pair consisting of `'QuasiSequence'` and a string or character vector with one of the following values:

- `"sobol"` — Quasi-random low-discrepancy sequences that use a base of two to form successively finer uniform partitions of the unit interval and then reorder the coordinates in each dimension

Note If `MonteCarloMethod` option is not specified or specified as`"standard"`, `QuasiSequence` is ignored.

Data Types: `string` | `char`

`BrownianMotionMethod` — Brownian motion construction method
`"standard"` (default) | string with value `"brownian-bridge"` or `"principal-components"` | character vector with value `'brownian-bridge'` or `'principal-components'`

Brownian motion construction method, specified as the comma-separated pair consisting of `'BrownianMotionMethod'` and a string or character vector with one of the following values:

- `"standard"` — The Brownian motion path is found by taking the cumulative sum of the Gaussian variates.
- `"brownian-bridge"` — The last step of the Brownian motion path is calculated first, followed by any order between steps until all steps have been determined.
- `"principal-components"` — The Brownian motion path is calculated by minimizing the approximation error.

The starting point for a Monte Carlo simulation is the construction of a Brownian motion sample path (or Wiener path). Such paths are built from a set of independent Gaussian variates, using either standard discretization, Brownian-bridge construction, or principal components construction.

Both standard discretization and Brownian-bridge construction share the same variance and therefore the same resulting convergence when used with the `MonteCarloMethod` using pseudo random numbers. However, the performance differs between the two when the `MonteCarloMethod` option `"quasi"` is introduced, with faster convergence seen for `"brownian-bridge"` construction option and the fastest convergence when using the `"principal-components"` construction option.

Data Types: `string` | `char`

`Processes` — Sequence of end-of-period processes or state vector adjustments
`simByTransition` makes no adjustments and performs no processing (default) | function | cell array of functions

Sequence of end-of-period processes or state vector adjustments, specified as the comma-separated pair consisting of `'Processes'` and a function or cell array of functions of the form

$$X_t = P(t, X_t)$$

`simByTransition` applies processing functions at the end of each observation period. The processing functions accept the current observation time t and the current state vector X_t, and return a state vector that might adjust the input state.

If you specify more than one processing function, `simByTransition` invokes the functions in the order in which they appear in the cell array.

Data Types: `cell` | `function`

Output Arguments

`Paths` — Simulated paths of correlated state variables

array

Simulated paths of correlated state variables, returned as an `(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`, `simByTransition` returns `Paths` as an empty matrix.

`Times` — Observation times associated with the simulated paths

column vector

Observation times associated with the simulated paths, returned as an `(NPeriods + 1)`-by-`1` column vector. Each element of `Times` is associated with the corresponding row of `Paths`.

More About

Transition Density Simulation

The CIR SDE has no solution such that $r(t) = f(r(0),\cdots)$.

In other words, the equation is not explicitly solvable. However, the transition density for the process is known.

The exact simulation for the distribution of $r(t\_1\ ),\cdots,r(t\_n)$ is that of the process at times $t\_1,\cdots,t\_n$ for the same value of $r(0)$. The transition density for this process is known and is expressed as

$$r(t) = \frac{\sigma^2(1 - e^{-\alpha(t - u)}}{4\alpha} x_d^2 \left(\frac{4\alpha e^{-\alpha(t - u)}}{\sigma^2(1 - e^{-\alpha(t - u)})} r(u) \right), t > u$$

where

$$d \equiv \frac{4b\alpha}{\sigma^2}$$

Bates Model

Bates models are bivariate composite models.

Each Bates model consists of two coupled univariate models:

- A geometric Brownian motion (`gbm`) model with a stochastic volatility function and jumps.

$$dX_{1t} = B(t)X_{1t}dt + \sqrt{X_{2t}}X_{1t}dW_{1t} + Y(t)X_{1t}dN_t$$

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 Bates price process.

Version History

Introduced in R2020b

R2022a: Perform Quasi-Monte Carlo simulation

Behavior changed in R2022a

Perform Quasi-Monte Carlo simulation using the name-value arguments `MonteCarloMethod` and `QuasiSequence`.

R2022b: Perform Brownian bridge and principal components construction

Behavior changed in R2022b

Perform Brownian bridge and principal components construction using the name-value argument `BrownianMotionMethod`.

References

[1] Glasserman, Paul *Monte Carlo Methods in Financial Engineering.* New York: Springer-Verlag, 2004.

[2] Van Haastrecht, Alexander, and Antoon Pelsser. "Efficient, Almost Exact Simulation of the Heston Stochastic Volatility Model." *International Journal of Theoretical and Applied Finance.* 13, no. 01 (2010): 1–43.

See Also

`bates` | `simByEuler` | `simByQuadExp`

Topics

“SDEs” on page 14-2
“SDE Models” on page 14-7
“SDE Class Hierarchy” on page 14-5
“Quasi-Monte Carlo Simulation” on page 14-62

simByTransition

Simulate Heston sample paths with transition density

Syntax

```
[Paths,Times] = simByTransition(MDL,NPeriods)
[Paths,Times] = simByTransition( ___ ,Name,Value)
```

Description

`[Paths,Times] = simByTransition(MDL,NPeriods)` simulates `NTrials` sample paths of Heston bivariate models driven by two Brownian motion sources of risk. `simByTransition` approximates continuous-time stochastic processes by the transition density.

`[Paths,Times] = simByTransition( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

You can perform quasi-Monte Carlo simulations using the name-value arguments for `MonteCarloMethod`, `QuasiSequence`, and `BrownianMotionMethod`. For more information, see “Quasi-Monte Carlo Simulation” on page 14-62.

Examples

Use simByTransition with heston Object

Simulate Heston sample paths with transition density.

Define the parameters for the `heston` object.

```
Return = 0.03;
Level = 0.05;
Speed = 1.0;
Volatility = 0.2;

AssetPrice = 80;
V0 = 0.04;
Rho = -0.7;
StartState = [AssetPrice;V0];
Correlation = [1 Rho;Rho 1];
```

Create a `heston` object.

```
hestonObj = heston(Return,Speed,Level,Volatility,'startstate',StartState,'correlation',Correlatio
```

```
hestonObj = 
   Class HESTON: Heston Bivariate Stochastic Volatility
   ----------------------------------------------------
     Dimensions: State = 2, Brownian = 2
   ----------------------------------------------------
      StartTime: 0
     StartState: 2x1 double array
```

```
    Correlation: 2x2 double array
          Drift: drift rate function F(t,X(t))
      Diffusion: diffusion rate function G(t,X(t))
     Simulation: simulation method/function simByEuler
         Return: 0.03
          Speed: 1
          Level: 0.05
     Volatility: 0.2
```

Define the simulation parameters.

```
nPeriods = 5;   % Simulate sample paths over the next five years
Paths = simByTransition(hestonObj,nPeriods);
Paths
```

```
Paths = 6×2

   80.0000    0.0400
   99.7718    0.0201
  124.7044    0.0176
   52.7914    0.1806
   75.3173    0.1732
   76.7572    0.1169
```

Quasi-Monte Carlo Simulation Using Heston Model

This example shows how to use `simByTransition` with a Heston model to perform a quasi-Monte Carlo simulation. Quasi-Monte Carlo simulation is a Monte Carlo simulation that uses quasi-random sequences instead pseudo random numbers.

Define the parameters for the `heston` object.

```
Return = 0.03;
Level = 0.05;
Speed = 1.0;
Volatility = 0.2;

AssetPrice = 80;
V0 = 0.04;
Rho = -0.7;
StartState = [AssetPrice;V0];
Correlation = [1 Rho;Rho 1];
```

Create a `heston` object.

```
Heston = heston(Return,Speed,Level,Volatility,'startstate',StartState,'correlation',Correlation)
```

```
Heston = 
   Class HESTON: Heston Bivariate Stochastic Volatility
   ----------------------------------------------------
     Dimensions: State = 2, Brownian = 2
   ----------------------------------------------------
      StartTime: 0
     StartState: 2x1 double array
    Correlation: 2x2 double array
```

```
       Drift: drift rate function F(t,X(t))
   Diffusion: diffusion rate function G(t,X(t))
  Simulation: simulation method/function simByEuler
      Return: 0.03
       Speed: 1
       Level: 0.05
  Volatility: 0.2
```

Perform a quasi-Monte Carlo simulation by using `simByTransition` with the optional name-value argument for `'MonteCarloMethod'`, `'QuasiSequence'`, and `'BrownianMotionMethod'`.

```
[paths,time] = simByTransition(Heston,10,'ntrials',4096,'MonteCarloMethod','quasi','QuasiSequence
```

Input Arguments

`MDL` — Stochastic differential equation model
`heston` object

Stochastic differential equation model, specified as a `heston` object. For more information on creating a `heston` object, see `heston`.

Data Types: `object`

`NPeriods` — Number of simulation periods
positive scalar integer

Number of simulation periods, specified as a positive scalar integer. The value of `NPeriods` determines the number of rows of the simulated output series.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[Paths,Times] = simByTransition(Heston,NPeriods,'DeltaTimes',dt)`

`NTrials` — Simulated trials (sample paths)
`1` (single path of correlated state variables) (default) | positive integer

Simulated trials (sample paths) of `NPeriods` observations each, specified as the comma-separated pair consisting of `'NTrials'` and a positive scalar integer.

Data Types: `double`

`DeltaTimes` — Positive time increments between observations
`1` (default) | scalar | column vector

Positive time increments between observations, specified as the comma-separated pair consisting of `'DeltaTimes'` and a scalar or `NPeriods`-by-`1` column vector.

`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.

Data Types: `double`

`NSteps` — Number of intermediate time steps

`1` (indicating no intermediate evaluation) (default) | positive integer

Number of intermediate time steps within each time increment *dt* (defined as `DeltaTimes`), specified as the comma-separated pair consisting of `'NSteps'` and a positive scalar integer.

The `simByTransition` function 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 `simByTransition` does not report the output state vector at these intermediate points, the refinement improves accuracy by enabling the simulation to more closely approximate the underlying continuous-time process.

Data Types: `double`

`MonteCarloMethod` — Monte Carlo method to simulate stochastic processes

`"standard"` (default) | string with values `"standard"`, `"quasi"`, or `"randomized-quasi"` | character vector with values `'standard'`, `'quasi'`, or `'randomized-quasi'`

Monte Carlo method to simulate stochastic processes, specified as the comma-separated pair consisting of `'MonteCarloMethod'` and a string or character vector with one of the following values:

- `"standard"` — Monte Carlo using pseudo random numbers.
- `"quasi"` — Quasi-Monte Carlo using low-discrepancy sequences.
- `"randomized-quasi"` — Randomized quasi-Monte Carlo.

Data Types: `string` | `char`

`QuasiSequence` — Low discrepancy sequence to drive the stochastic processes

`"sobol"` (default) | string with value `"sobol"` | character vector with value `'sobol'`

Low discrepancy sequence to drive the stochastic processes, specified as the comma-separated pair consisting of `'QuasiSequence'` and a string or character vector with one of the following values:

- `"sobol"` — Quasi-random low-discrepancy sequences that use a base of two to form successively finer uniform partitions of the unit interval and then reorder the coordinates in each dimension

Note If `MonteCarloMethod` option is not specified or specified as`"standard"`, `QuasiSequence` is ignored.

Data Types: `string` | `char`

`BrownianMotionMethod` — Brownian motion construction method

`"standard"` (default) | string with value `"brownian-bridge"` or `"principal-components"` | character vector with value `'brownian-bridge'` or `'principal-components'`

Brownian motion construction method, specified as the comma-separated pair consisting of `'BrownianMotionMethod'` and a string or character vector with one of the following values:

- `"standard"` — The Brownian motion path is found by taking the cumulative sum of the Gaussian variates.

- "brownian-bridge" — The last step of the Brownian motion path is calculated first, followed by any order between steps until all steps have been determined.
- "principal-components" — The Brownian motion path is calculated by minimizing the approximation error.

The starting point for a Monte Carlo simulation is the construction of a Brownian motion sample path (or Wiener path). Such paths are built from a set of independent Gaussian variates, using either standard discretization, Brownian-bridge construction, or principal components construction.

Both standard discretization and Brownian-bridge construction share the same variance and therefore the same resulting convergence when used with the MonteCarloMethod using pseudo random numbers. However, the performance differs between the two when the MonteCarloMethod option "quasi" is introduced, with faster convergence seen for "brownian-bridge" construction option and the fastest convergence when using the "principal-components" construction option.

Data Types: string | char

StorePaths — Flag for storage and return method
True (default) | logical with True or False

Flag for storage and return method that indicates how the output array Paths is stored and returned, specified as the comma-separated pair consisting of 'StorePaths' and a scalar logical flag with a value of True or False.

- If StorePaths is True (the default value) or is unspecified, then simByTransition returns Paths as a three-dimensional time series array.
- If StorePaths is False (logical 0), then simByTransition returns the Paths output array as an empty matrix.

Data Types: logical

Processes — Sequence of end-of-period processes or state vector adjustments
simByTransition makes no adjustments and performs no processing (default) | function | cell array of functions

Sequence of end-of-period processes or state vector adjustments, specified as the comma-separated pair consisting of 'Processes' and a function or cell array of functions of the form

$$X_t = P(t, X_t)$$

simByTransition applies processing functions at the end of each observation period. The processing functions accept the current observation time t and the current state vector X_t, and return a state vector that might adjust the input state.

If you specify more than one processing function, simByTransition invokes the functions in the order in which they appear in the cell array.

Data Types: cell | function

Output Arguments

Paths — Simulated paths of correlated state variables
array

Simulated paths of correlated state variables, returned as an `(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`, `simByTransition` returns `Paths` as an empty matrix.

`Times` — Observation times associated with simulated paths
column vector

Observation times associated with the simulated paths, returned as an `(NPeriods + 1)`-by-`1` column vector. Each element of `Times` is associated with the corresponding row of `Paths`.

More About

Transition Density Simulation

The CIR SDE has no solution such that $r(t) = f(r(0),\cdots)$.

In other words, the equation is not explicitly solvable. However, the transition density for the process is known.

The exact simulation for the distribution of *r*(*t*_1),···,*r*(*t*_*n*) is that of the process at times *t*_1,···,*t*_*n* for the same value of $r(0)$. The transition density for this process is known and is expressed as

$$r(t) = \frac{\sigma^2(1 - e^{-\alpha(t-u)}}{4\alpha}x_d^2\left(\frac{4\alpha e^{-\alpha(t-u)}}{\sigma^2(1 - e^{-\alpha(t-u)})}r(u)\right), t > u$$

where

$$d \equiv \frac{4b\alpha}{\sigma^2}$$

Heston Model

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.

Version History

Introduced in R2020b

R2022a: Perform Quasi-Monte Carlo simulation
Behavior changed in R2022a

Perform Quasi-Monte Carlo simulation using the name-value arguments `MonteCarloMethod` and `QuasiSequence`.

R2022b: Perform Brownian bridge and principal components construction
Behavior changed in R2022b

Perform Brownian bridge and principal components construction using the name-value argument `BrownianMotionMethod`.

References

[1] Glasserman, Paul *Monte Carlo Methods in Financial Engineering.* New York: Springer-Verlag, 2004.

[2] Van Haastrecht, Alexander, and Antoon Pelsser. "Efficient, Almost Exact Simulation of the Heston Stochastic Volatility Model." *International Journal of Theoretical and Applied Finance.* 13, no. 01 (2010): 1–43.

See Also

`simByEuler` | `simByQuadExp` | `heston`

Topics

“SDEs” on page 14-2
“SDE Models” on page 14-7
“SDE Class Hierarchy” on page 14-5
“Quasi-Monte Carlo Simulation” on page 14-62

A

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, Irwin Professional 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." https://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 `https://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, (Sept., 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.