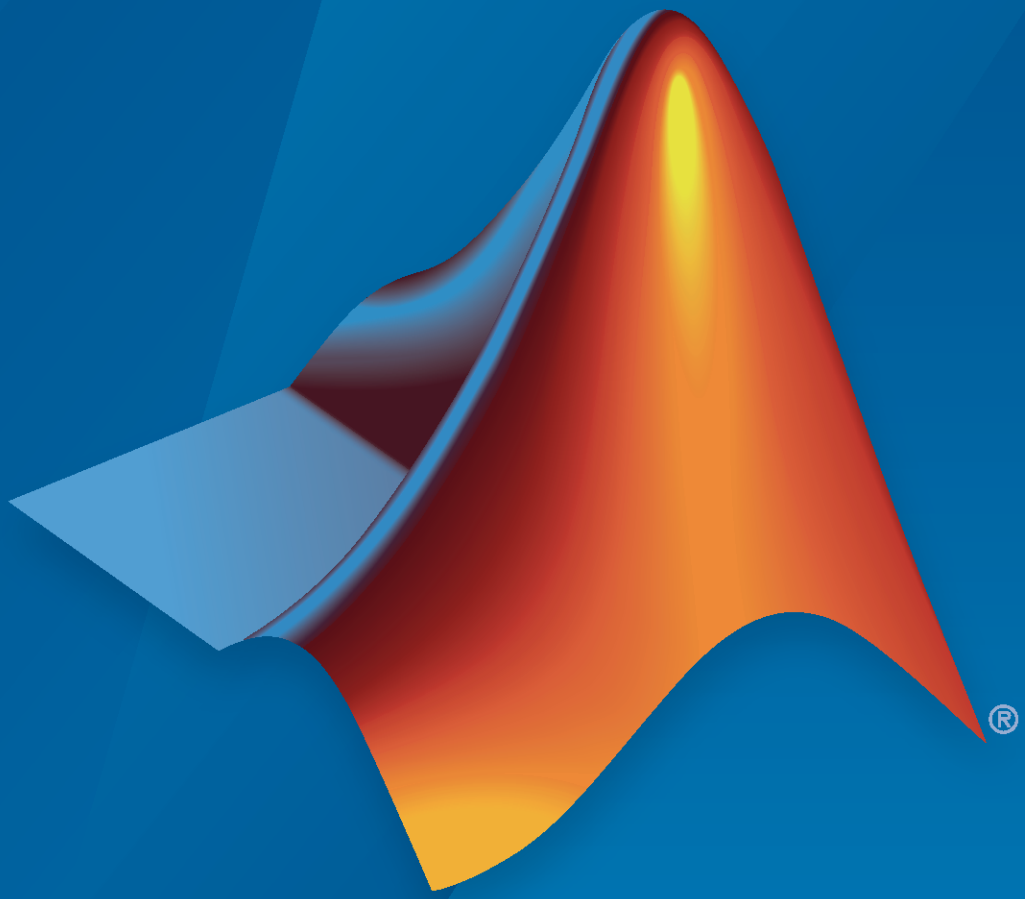
Financial Toolbox™

User's Guide



MATLAB®

R2025b

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

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)
September 2023	Online only	Revised for Version 23.2 (R2023b)
March 2024	Online only	Revised for Version 24.1 (R2024a)
September 2024	Online only	Revised for Version 24.2 (R2024b)
March 2025	Online only	Revised for Version 25.1 (R2025a)
September 2025	Online only	Rereleased for Version 25.2 (R2025b)

Contents

1 Getting Started

2 Performing Common Financial Tasks

3 Portfolio Analysis

4 Mean-Variance Portfolio Optimization Tools

5 CVaR Portfolio Optimization Tools

6 MAD Portfolio Optimization Tools

7 Investment Performance Metrics

8 Credit Risk Analysis

9 Regression with Missing Data

10 Solving Sample Problems

11 Using Financial Timetables

12 Trading Date Utilities

13 Technical Analysis

14 Stochastic Differential Equations

15 Functions

A Bibliography

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

In this section...

Introduction

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

Key Definitions

Matrix

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

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

Vector

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

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

Scalar

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

Referencing Matrix Elements

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

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

```
ans =

          0.06
```

```
Cash(3)

ans =

       5280.00
```

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

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

adds another row to the matrix and creates

```
Bonds =

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

Likewise,

```
Prices = [987.50
          475.00
          995.00]

Bonds = [Prices, Bonds]
```

adds another column and creates

```
Bonds =

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

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

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

BondItems =

    500.00   0.055   4
```

Transposing Matrices

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

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

produces

```
Cash =

        1500
        4470
        5280
       -1299
```

See Also

More About

- “Matrix Algebra Refresher” on page 1-7
- “Using Input and Output Arguments with Functions” on page 1-15

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

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

then the dot product

$$X \cdot Y = x_1y_1 + x_2y_2 + ... + x_ny_n$$

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

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

```
Value =

      3.5413e+004
```

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

```
Value =

      3.5413e+004
```

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

Multiplying Vectors and Matrices

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

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

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

```
WeekPortValue = WeekClosePr * PortQuan
```

```
WeekPortValue =

1.0e+004 *

    3.5412
    3.5587
```

```
    3.5475
    3.5550
    3.5513
```

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

Multiplying Two Matrices

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

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

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

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

To illustrate, assume that there are two portfolios of the same three stocks 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 $(A*X \neq X*A)$, different processes apply. In formal matrix algebra, the solution involves matrix inversion. MATLAB, however, simplifies the process by providing two matrix division symbols, left and right (`\` and `/`). In general,

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

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

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

Solving Simultaneous Linear Equations

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

A linear equation is any equation of the form

$$a_1x + a_2y = b,$$

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

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

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

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, MATLAB software 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

This example shows how to compute the internal rate of return of the cash stream using `irr`.

Specify the income stream as 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];
```

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

```
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

This example shows how to compute the present or future value of cash flows at regular or irregular time intervals with equal or unequal payments.

To compute the present or future value, you can use the following functuions: `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.

Specify the income stream as 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];
```

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

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

```
ROR =
0.1172
```

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

This example shows how to compute standard depreciation schedules using `depgendb`.

The following code 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).

```
Decline1 = depgendb(15000, 1500, 5, 1.5)
```

```
Decline1 = 1×5
10^3 ×

    4.5000    3.1500    2.2050    1.5435    2.1015
```

```
Decline2 = depgendb(15000, 1500, 5, 2.0)
```

```
Decline2 = 1×5
10^3 ×

    6.0000    3.6000    2.1600    1.2960    0.4440
```

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

Annuities

This example shows how to work with annuities using `annurate`.

The following code 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.

You can use a present-value function (`pvfix`) 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.

You can compute an amortization schedule using `amortize` for a loan or annuity. For example, 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)
```

```
Prpmt = 1×12

  399.7574  402.7556  405.7762  408.8196  411.8857  414.9748  418.0872  421.2228  424.3820  427.5

Intpmt = 1×12

   37.5000   34.5018   31.4812   28.4378   25.3717   22.2825   19.1702   16.0346   12.8754    9.6

Balance = 1×12
10^3 ×

    4.6002    4.1975    3.7917    3.3829    2.9710    2.5560    2.1379    1.7167    1.2923    0.8

Payment =
437.2574
```

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`.

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. For more information on `Basis`, see EMU and Market Conventions: Recent Developments.

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.
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.

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 to compute the price of a bond with an odd first period using `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.

Note, `bndprice` uses 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.

Yield Functions

This example shows how to use `bndyield` compute the yield of a bond that has odd first and last periods and settlement in the first period.

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;
```

Call 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

This example shows how to compute the annualized Macaulay and modified durations and the periodic Macaulay duration for a bond.

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 code 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, 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 = datetime(2000,1,12);
Maturity = datetime(2001,10,1);

[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

This example shows how to compute the key rate duration of the US Treasury Bond.

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.

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 using `bndkrdur` 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 = 1×4

    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 = 1×2

   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

This 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
```

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`.

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 following 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

This example shows how to convert a Treasury bill discount to an equivalent yield.

You can examine the money-market and bond-equivalent yields of a 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)

BreakevenDiscount =
0.0167
```

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

BEYield = 2×1

    0.0164
    0.0170


MMYield = 2×1

    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

This example shows how to derive and analyze interest-rate curves, including data conversion and extrapolation, bootstrapping, and interest-rate curve conversions.

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. The following code uses only one security each; analysts often use 30, 100, or more of each.

First, capture Treasury bill quotes and Reasury bond quotes in their reported format.

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

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

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

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

Use the `tbl2bond` to convert the Treasury bill data to Treasury bond format.

```
TBTBond = tbl2bond(TBill)
```

```
TBTBond = 1×5
10^5 ×

         0    7.3085    0.0010    0.0010    0.0000
```

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

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

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

```
TBondsAll = 2×5
10^5 ×

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

Use `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 = 2×6
10^5 ×
```

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


Prices = 2×1

   99.2654
  103.1875


Yields = 2×1

    0.0510
    0.0564
```

Use a bootstrapping function 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 = 2×1

    0.0516
    0.0558


CurveDates = 2×1

      730846
      731160
```

`CurveDates` gives the investment horizon.

```
datestr(CurveDates)

ans = 2×11 char array
    '26-Dec-2000'
    '05-Nov-2001'
```

Use the 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)

DiscRates = 2×1

    0.9926
    0.9462
```

```
CurveDates = 2×1

      730846
      731160


[FwdRates, CurveDates] = zero2fwd(ZeroRates, CurveDates, Settle)

FwdRates = 2×1

    0.0516
    0.0565


CurveDates = 2×1

      730846
      731160


[PYldRates, CurveDates] = zero2pyld(ZeroRates, CurveDates, Settle)

PYldRates = 2×1

    0.0522
    0.0557


CurveDates = 2×1

      730846
      731160
```

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 Δt in the return formulas and they multiply by Δt in the price formulas. A positive factor of Δt (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: $p_s(t-1) \to 0$) 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 $\pm\pi$, 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	$0.0000-\pi i$

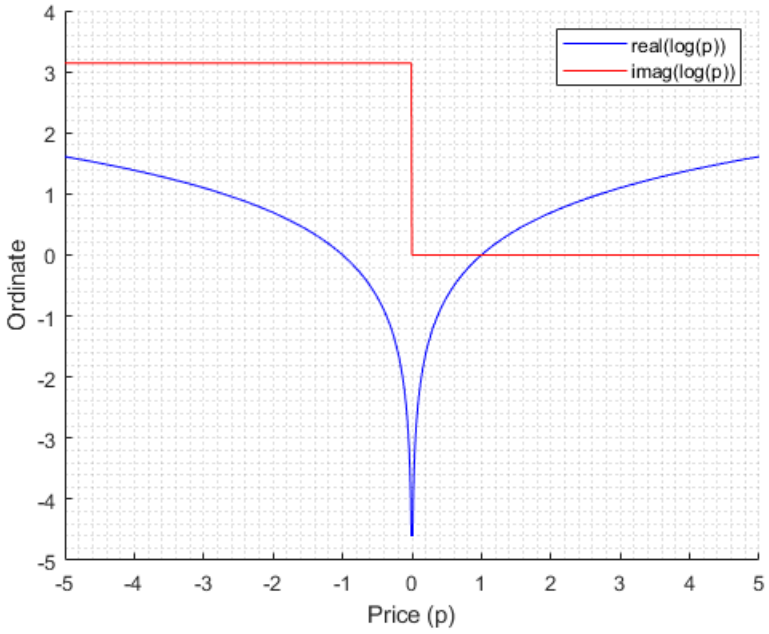
The real part shows the trend in the absolute price series. When $|p_c(t-1)| < |p_c(t)|$, that is, when prices move away from zero, $r_c(t)$ has a positive real part. When $|p_c(t-1)| > |p_c(t)|$, that is, when prices move toward zero, $r_c(t)$ has a negative real part. When $|p_c(t-1)| = |p_c(t)|$, 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
```



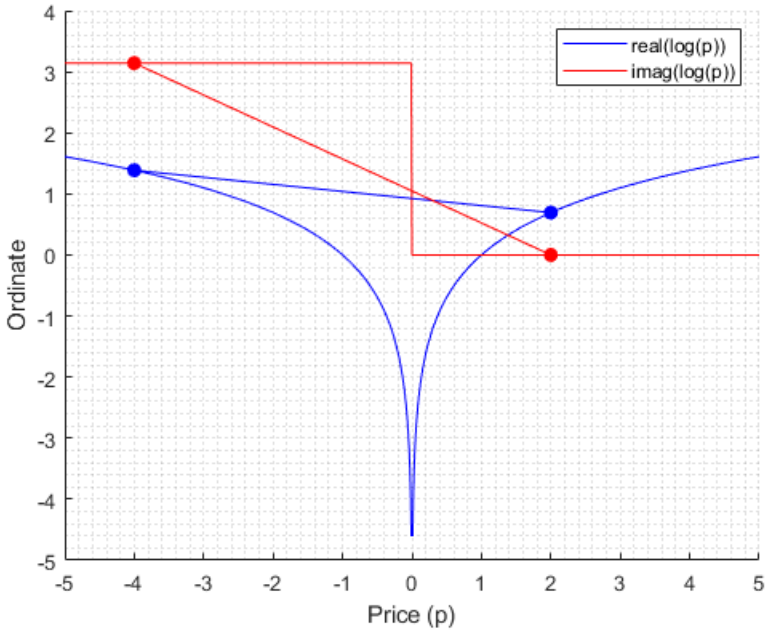
Due to the following identity

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

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 of 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

This example shows how to compute the call and put prices of a European option and its delta, gamma, lambda, and implied volatility.

Using the Black-Scholes model entails several assumptions:

- The prices of the underlying asset follow an Ito process. (See “Derivatives Pricing and Yields” 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 (`blsprice`, `blsdelta`, `blsgamma`, `blsvega`, and `blslambda`) 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.

```
[OptCall, OptPut] = blsprice(100, 95, 0.10, 0.25, 0.50, 0)
```

```
OptCall =
13.6953

OptPut =
6.3497
```

```
[CallVal, PutVal] = blsdelta(100, 95, 0.10, 0.25, 0.50, 0)
```

```
CallVal =
0.6665

PutVal =
-0.3335
```

```
GammaVal = blsgamma(100, 95, 0.10, 0.25, 0.50, 0)
```

```
GammaVal =
0.0145
```

```
VegaVal = blsvega(100, 95, 0.10, 0.25, 0.50, 0)
```

```
VegaVal =
18.1843
```

```
[LamCall, LamPut] = blslambda(100, 95, 0.10, 0.25, 0.50, 0)
```

```
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`

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

This example shows how to price an American call option using a 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. 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. The function `binprice` 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).

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

StockPrice = 6×6

  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


OptionPrice = 6×6

   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

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

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

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

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

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

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

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

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);
```

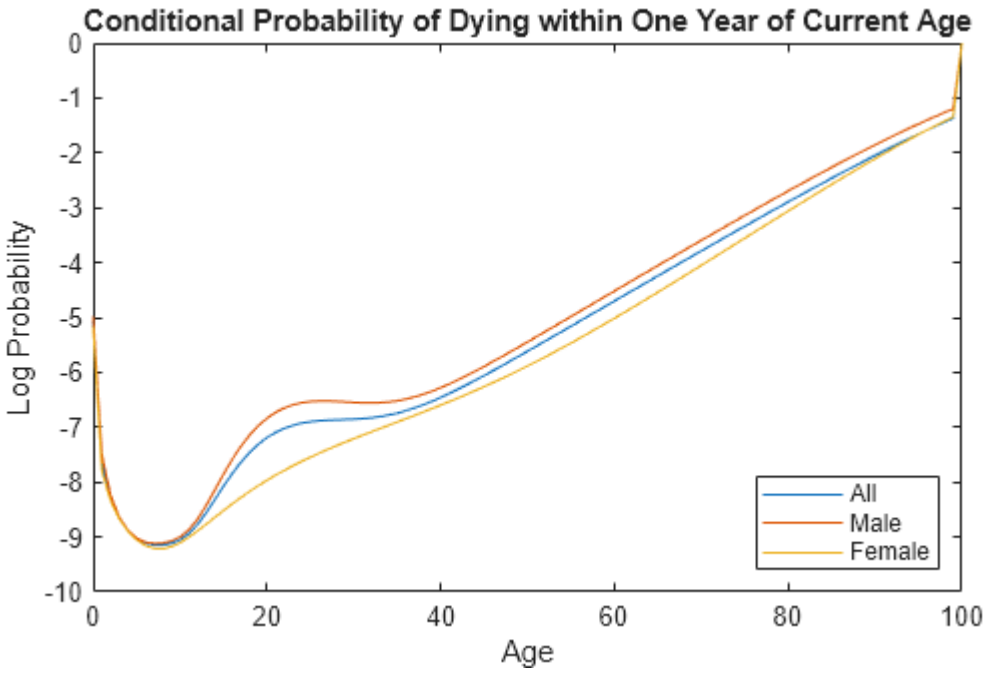
Generate life table series from the calibrated mortality model.

```
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');
```



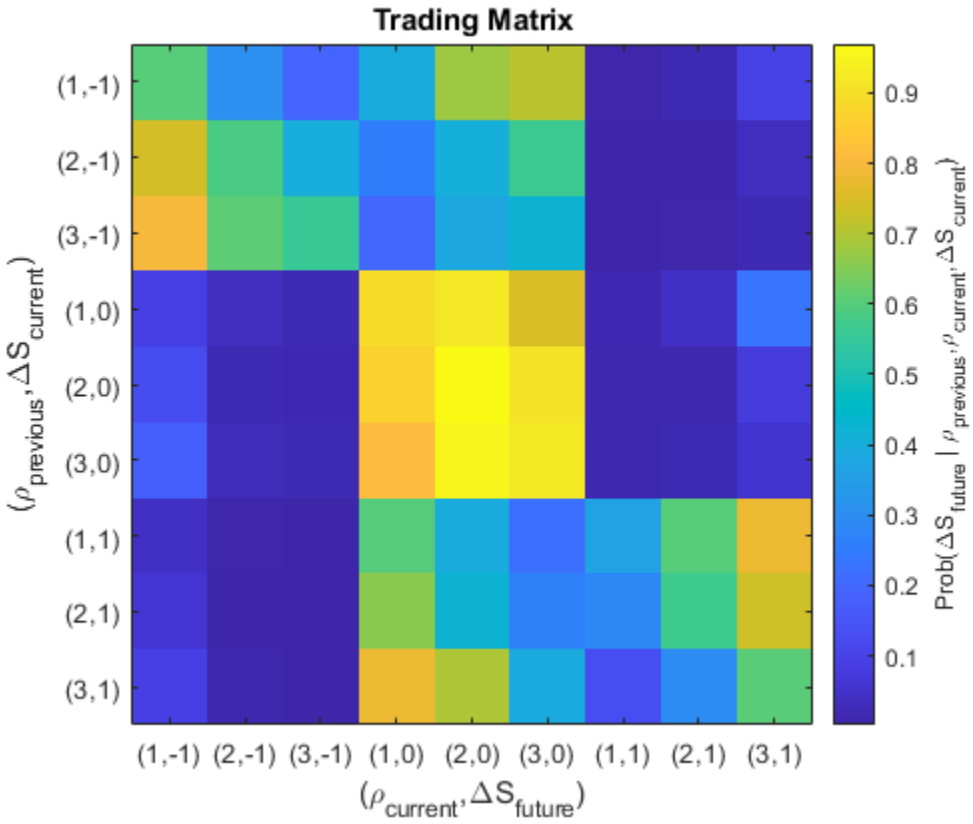
See Also

`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 2-59
3. “Machine Learning for Statistical Arbitrage III: Training, Tuning, and Prediction” on page 2-69

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™ documentation 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 argument to `false`. Replace the headers with descriptive variable names obtained from

`LOBSTER_SampleFiles_ReadMe.txt`. Set the `'ReadSize'` name-value 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 =

  M×5 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 =

  M×4 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             4899  table
```

```
  DT                Mx13            5292  tall
  DTPreview         Mx5             2926  tall
  DTT               Mx12            5056  tall
  DTTPreview        Mx4             2704  tall
  LOB3Variables     1x12             952  string
  LOBFileName       1x1              262  string
  LOBPreview        8x5             2331  table
  MSGFileName       1x1              246  string
  TimeVariable      1x1              166  string
  date              1x1              182  string
  rem               1x1              246  string
  ticker            1x1              166  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 4.5 sec
Evaluation completed in 5 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 =

  581,030×2 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\tp5b508469\finance-ex97702880\ExchangeData\INTC\2012-06-21
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 1: Completed in 5.8 sec
Evaluation completed in 6.7 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 4.1 sec
Evaluation completed in 4.2 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             5056  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 4.1 sec
Evaluation completed in 4.4 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 3.7 sec
- Pass 2 of 2: Completed in 4 sec
Evaluation completed in 8.6 sec
```

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

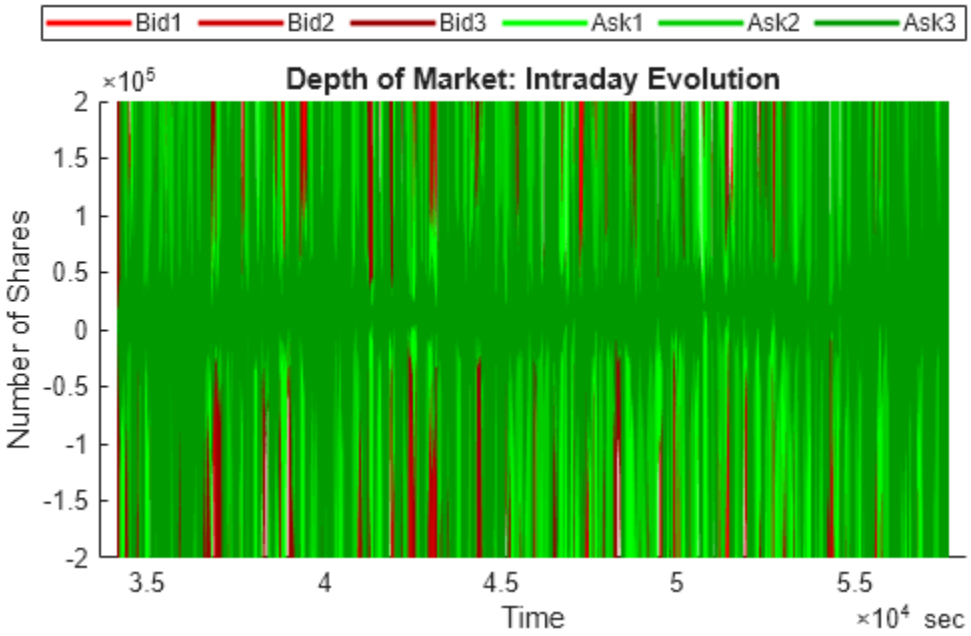
```
figure
hold on

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

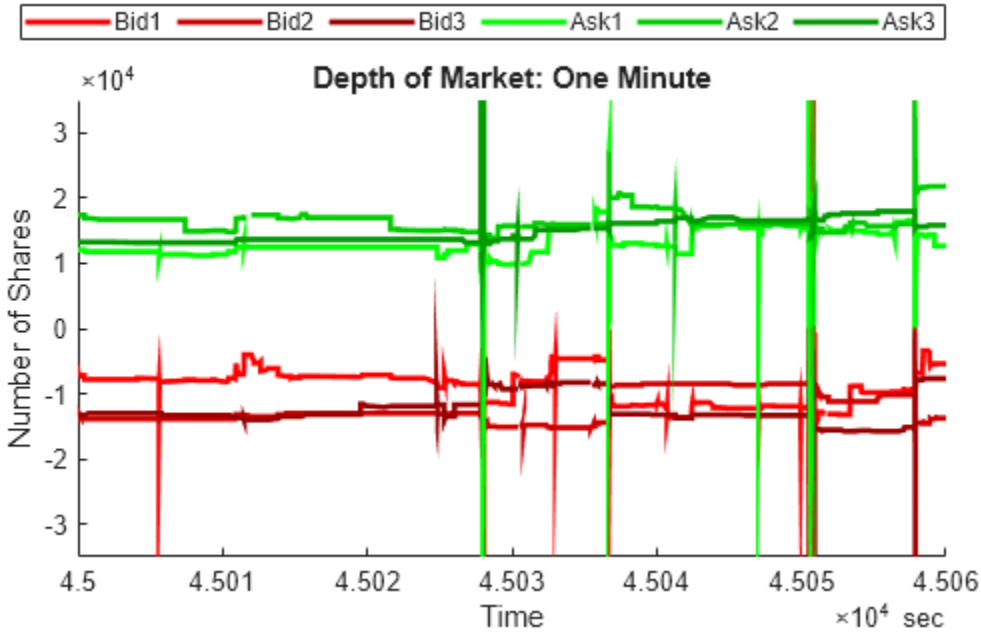
Some visualization functions work directly with tall arrays and do not require the use of `gather` (see "Visualization of Tall Arrays"). The functions automatically sample data to decrease pixel density. Visualize the level 3 intraday *depth of market*, which shows the time evolution of liquidity, by 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 2-59
- “Machine Learning for Statistical Arbitrage III: Training, Tuning, and Prediction” on page 2-69

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

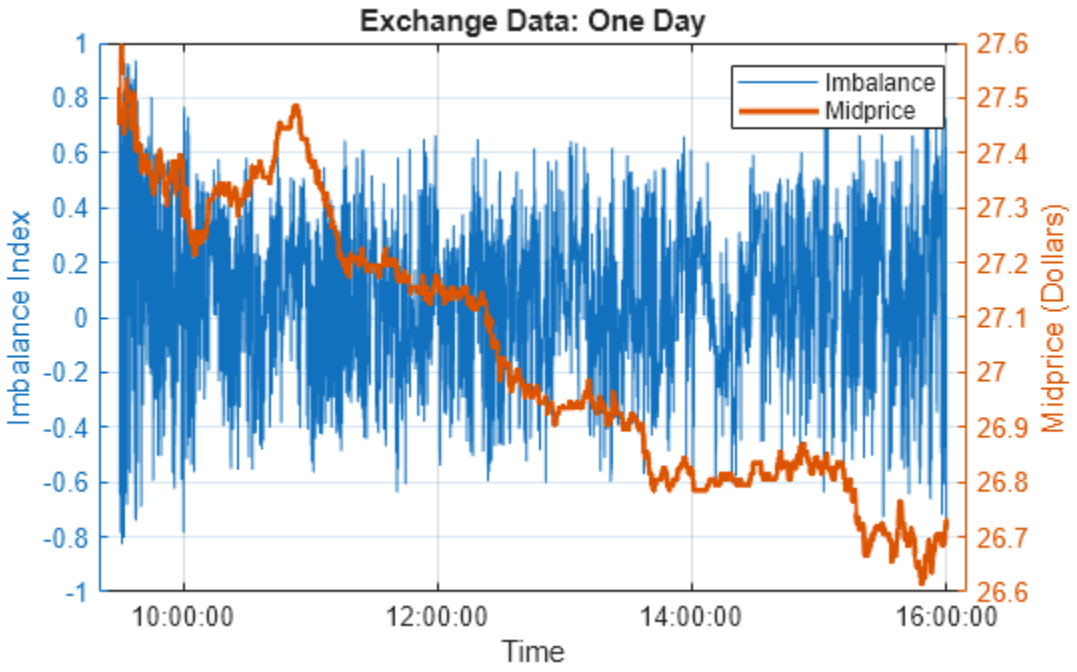
Load `LOBVars.mat`, the preprocessed LOB data set of the NASDAQ security INTC, which is included with the Financial Toolbox™ documentation.

```
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`.

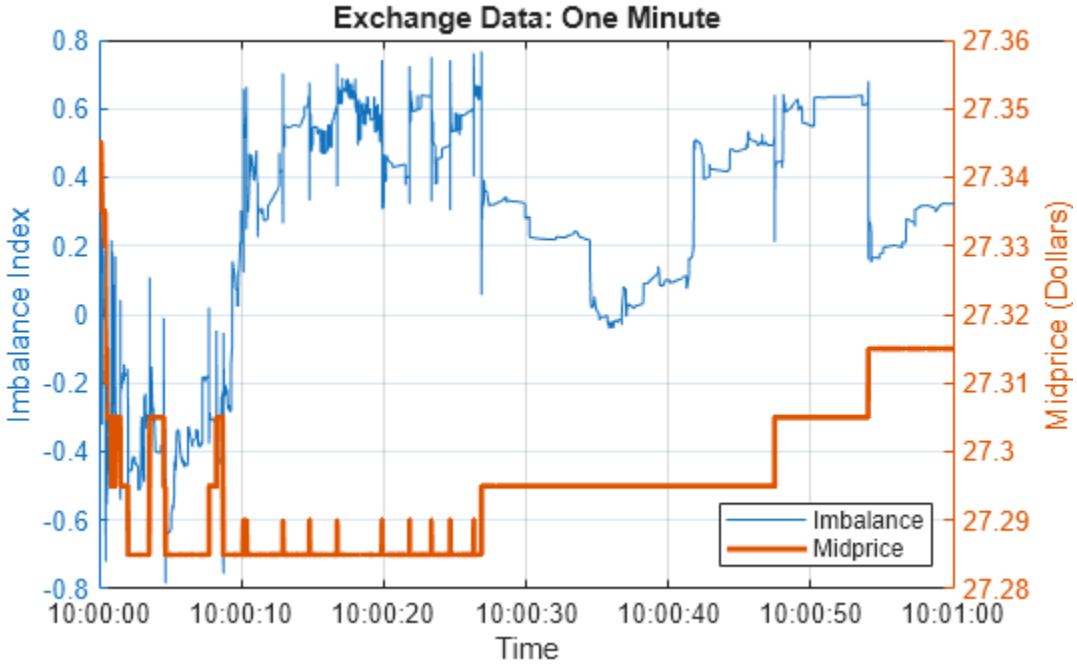
```
figure

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.

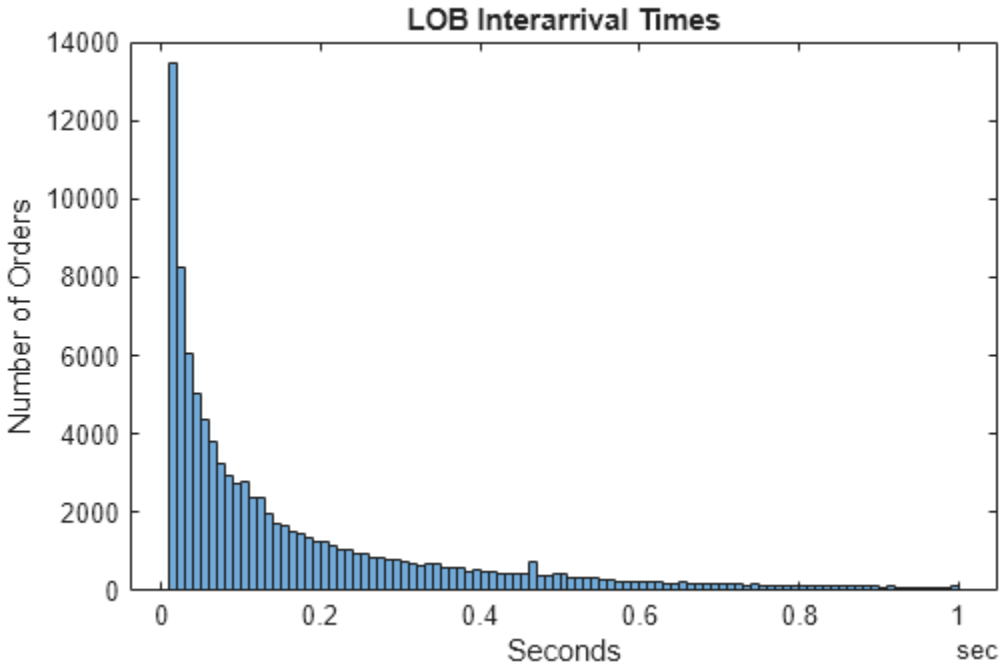
```
timeRange = seconds([36000 36060]); % One minute after 10 AM, when prices were climbing
xlim(timeRange)
legend('Location','SE')
title("Exchange Data: One Minute")
```



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.

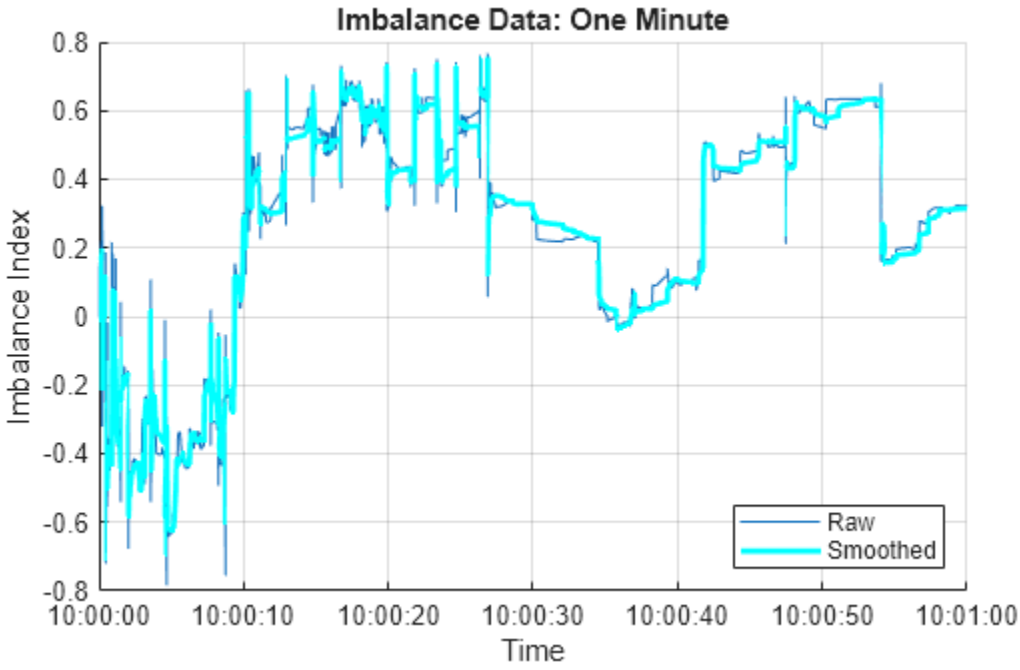
```
dI = 10; % Hyperparameter
dTI = dI*DTAvg

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
```

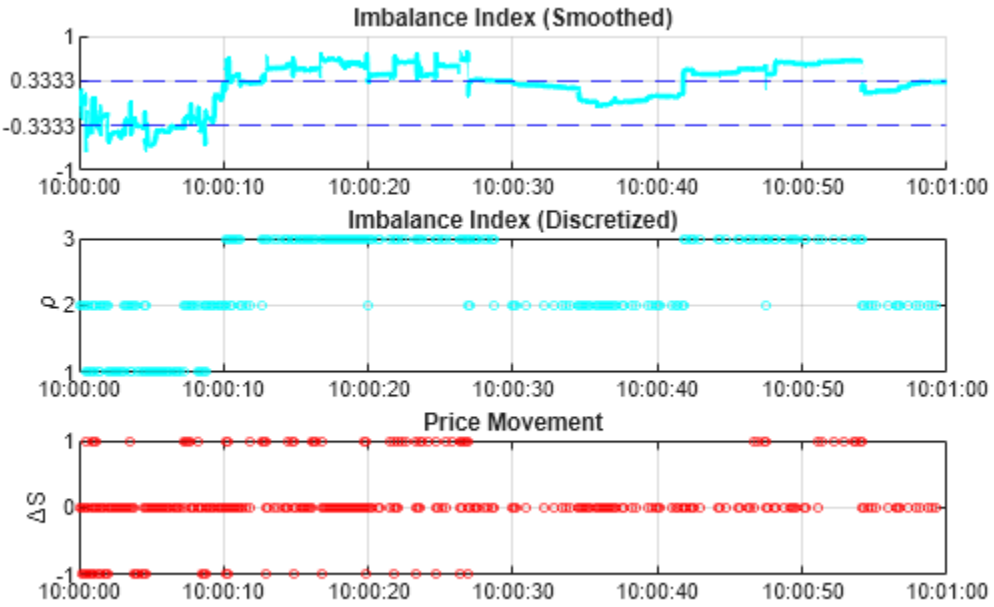
```
dTS = duration
   0.80547 sec
```

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
```



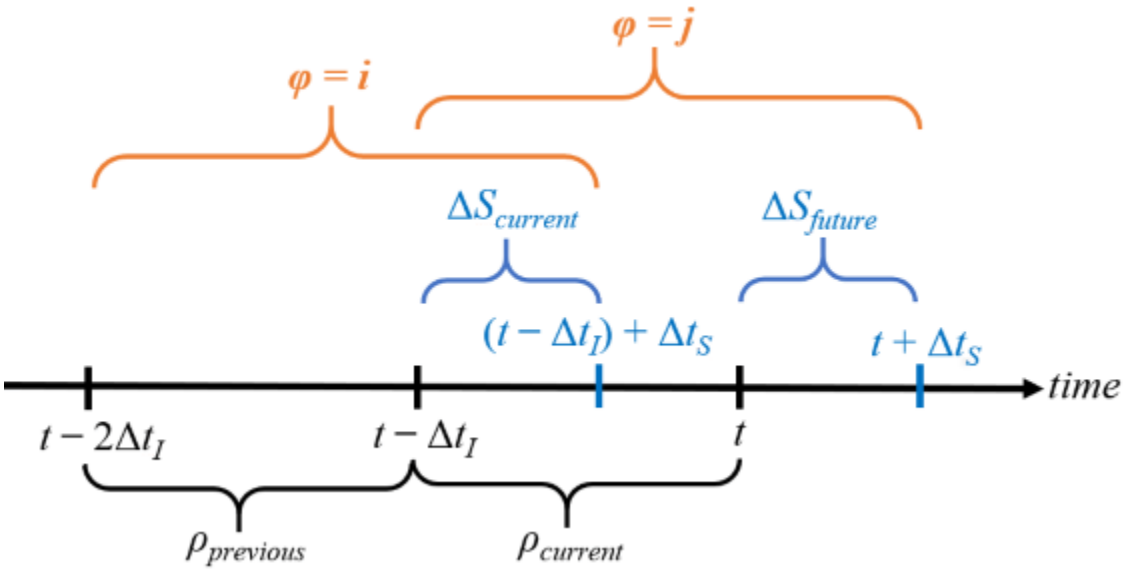
Continuous Time Markov Process

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 $\varphi = \left(\rho_{previous}, \Delta S_{current}\right) = i$ to $\varphi = \left(\rho_{current}, \Delta S_{future}\right) = j$ (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 $\text{Prob}\left(\Delta S_{future} \middle| \rho_{previous}, \rho_{current}, \Delta S_{current}\right)$ as in [4], apply Bayes' rule,

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

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 $\varphi = (\rho, \Delta S)$.

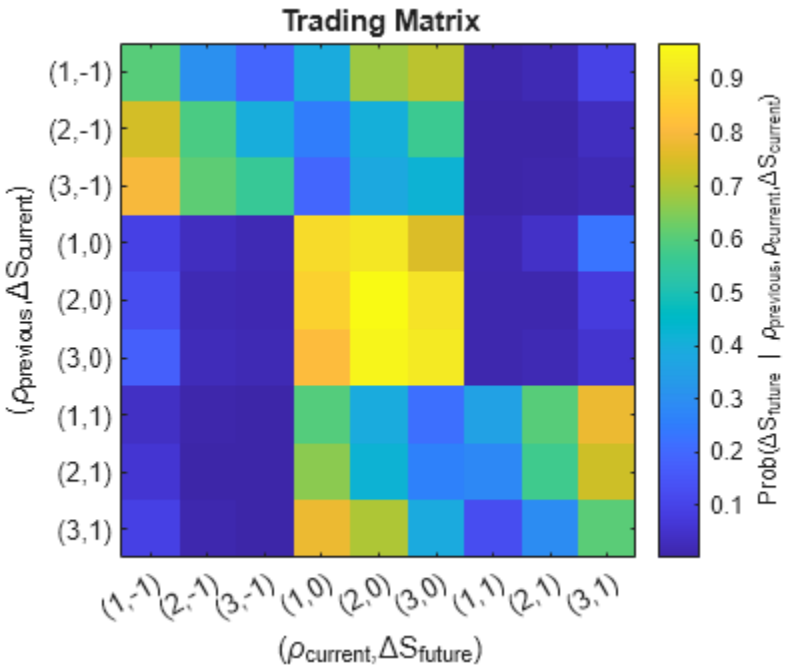
```
binNames = string(1:numBins);
stateNames = ["("+binNames+",-1)","("+binNames+",0)","("+binNames+",1)"];
QTable = array2table(Q,'RowNames',stateNames,'VariableNames',stateNames)
```

```
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 = 9×9 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 2-69.

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 2-69

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

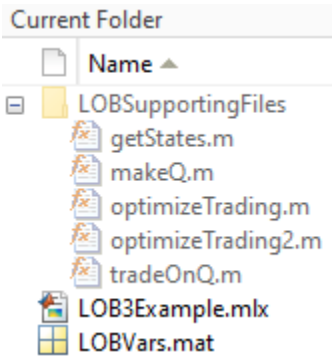
This example uses Bayesian optimization to tune hyperparameters in the algorithmic trading model, supervised by the end-of-day return. It is part of a series of related examples on machine learning for statistical arbitrage (see “Machine Learning Applications”).

Load `LOBVars.mat`, the preprocessed LOB data set of the NASDAQ security INTC, which is included with the Financial Toolbox™ documentation.

```
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.



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 a 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 = 9×9 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 principal advantages is the absence of costly derivative evaluations. To implement Bayesian optimization, use the Statistics and Machine Learning Toolbox™ function `bayesopt`.

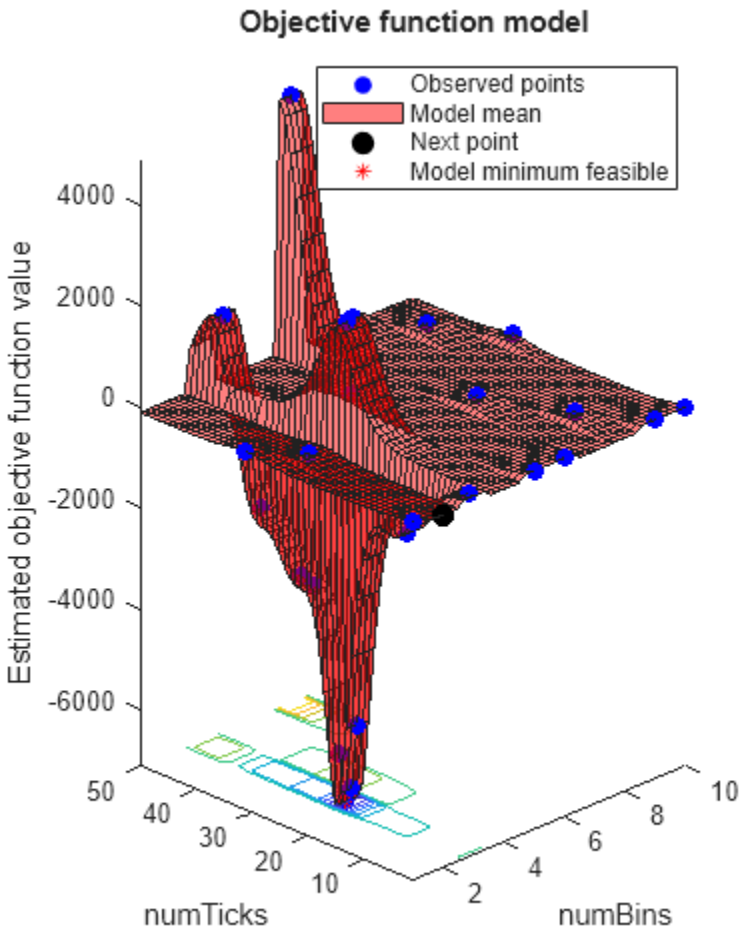
The supporting function `optimizeTrading.m` uses `bayesopt` to optimize the trading strategy in `tradeOnQ`.

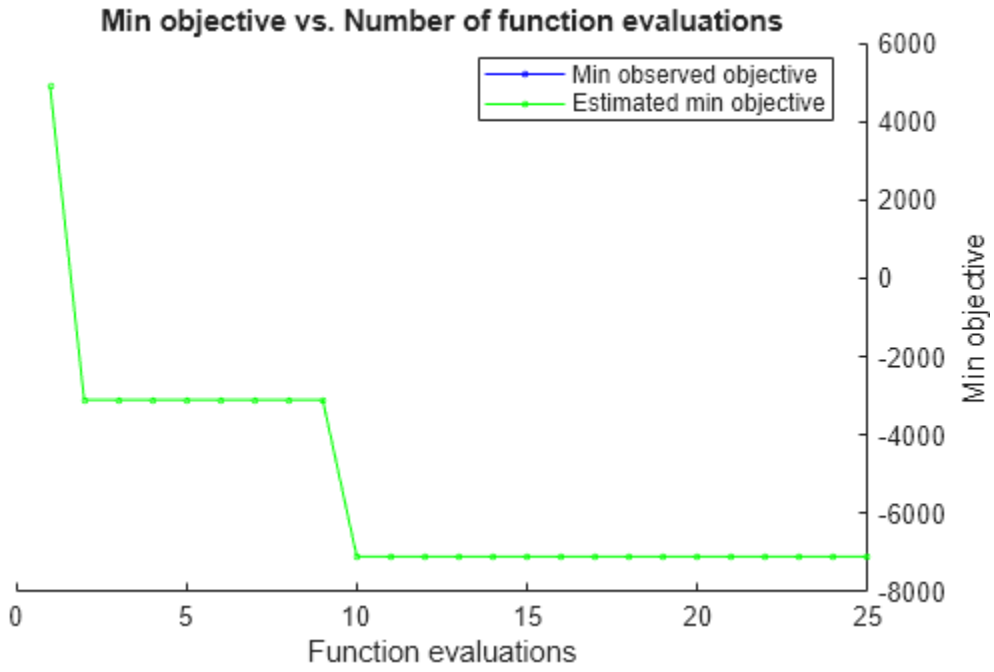
```
function results = optimizeTrading(TData,VData)

% 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`.

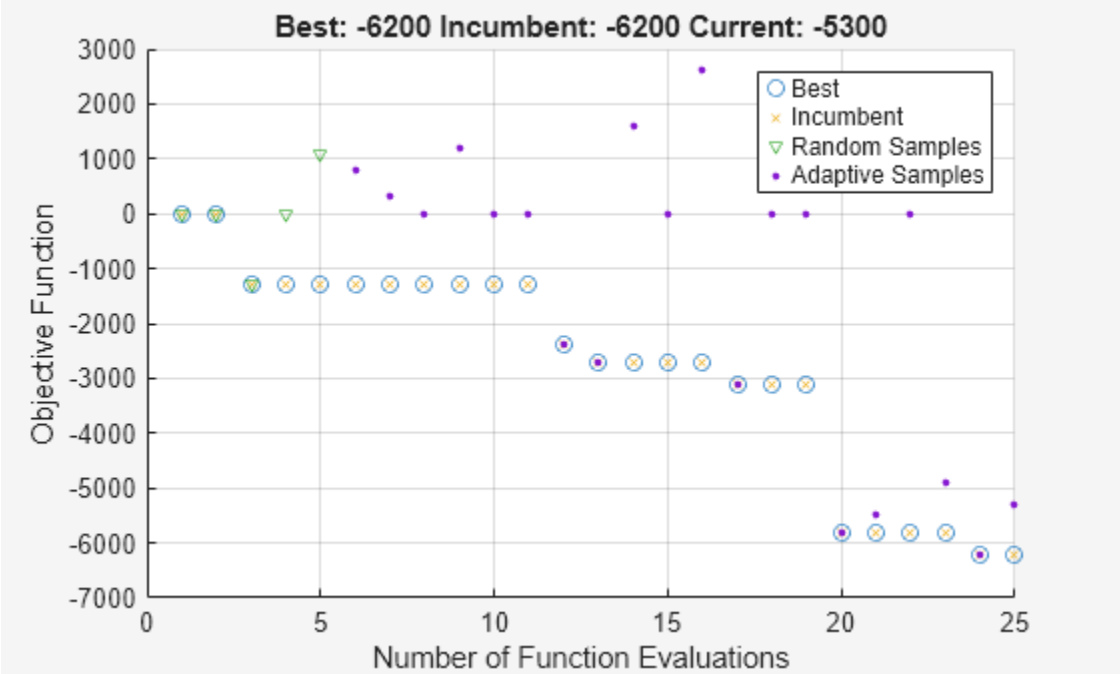
```
[Calibration,negReturn] = bestPoint(results,'Criterion','min-observed')
```

```
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    25
```

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 2-59
- “Backtest Deep Learning Model for Algorithmic Trading of Limit Order Book Data” on page 2-78
- “Deep Reinforcement Learning for Optimal Trade Execution” on page 4-407

Backtest Deep Learning Model for Algorithmic Trading of Limit Order Book Data

This example applies artificial intelligence (AI) techniques and backtesting to find trading opportunities from imbalances in limit order book (LOB) data of a security. Specifically, the example applies a backtest trading strategy to measure the performance of a long short-term memory (LSTM) neural network. The goal of the model is to predict future forward price movements from current and past on ask and bid volume imbalances and forward price movements, which are extracted from the raw LOB data. Given a forward price movement prediction, a trading decision can be made. This example uses the following trading strategy and assumptions:

- Trading strategy: "All-in/all-out"; all cash is invested when the model predicts positive price movements and cash out when the model predicts no or negative cash movements.
- Transactions are free.
- Current trading actions do not affect future price movements.
- The risk-free rate is 0%.

This example builds on the developments in the “Machine Learning for Statistical Arbitrage: Introduction” on page 2-48 example series and uses this procedure:

1. Load and preprocess the raw limit order book data.
2. Engineer features by transforming the raw LOB data to obtain the discretized imbalance index series ρ_t and forward price movement series ΔS_t. The response is the discretized future price movement $\Delta S_{t+\Delta t_S}$.
3. Prepare the data for the classification LSTM neural network.
4. Partition the data into training, validation, and backtest sets.
5. Configure the LSTM neural network.
6. Train and validate the LSTM neural network.
7. Asses the quality of the trained model.
8. Backtest the trained LSTM neural network. Compare the performance of the model with a model that has knowledge of all future price movements (maximum profit model).

Load and Preprocess Raw LOB Data

This example uses the level 3 limit order book data from one trading day of NASDAQ exchange data [3] on one security (INTC) in a sample provided by LOBSTER [2] and included with the Financial Toolbox™ documentation in the zip file `LOBSTER_SampleFile_INTC_2012-06-21_5.zip`.

Extract the contents of the zip file into your current folder.

```
unzip("LOBSTER_SampleFile_INTC_2012-06-21_5.zip");
MSGFileName = "INTC_2012-06-21_34200000_57600000_message_5.csv";
LOBFileName = "INTC_2012-06-21_34200000_57600000_orderbook_5.csv";
```

The “Machine Learning for Statistical Arbitrage I: Data Management and Visualization” on page 2-50 example describes LOB data. To summarize:

- LOB data is composed of two CSV files: the order book `LOBFileName` and a message file `MSGFileName`.

- The order book data describes the intraday evolution of the limit order book of the security, which includes market and limit orders, resulting buys and sells, and corresponding times of all events.
- The message file describes each event.
- Many order events can occur during a trading day, therefore such data sets are typically large and difficult or impossible to fit in memory. This example uses a `tabularTextDatastore` and tall timetables to store and operate on the sets.

Extract the trading date from the message file name.

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

Create separate datastores for the message and data files by using `tabularTextDatastore`. Ignore the generic column headers by setting `ReadVariableNames=false`. To allow similarly formatted files to be appended to existing datastores at the end of each trading day `ReadSize="file"`. Set descriptive variables names to each datastore.

```
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 the time variable of the message data and all levels in the limit order data.

```
DSMSG.SelectedVariableNames = ["Time" "Type"];
DSLOB.SelectedVariableNames = DSLOB.VariableNames;

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

Engineer Features and Prepare Response Variable

AI techniques can accommodate data in a variety of forms, from high-dimensional raw data, in which it's difficult for humans to find patterns and structure, through a lower dimensional space where the variables in the raw data are transformed to more practical, interpretable measurements. For example, Kolm, et al. 2021 [1] train a convolutional neural network (CNN) and LSTM on raw LOB data to forecast for trading opportunities. Regardless of form, with careful tuning, the layers of a deep network can find structure in the data that might inform predictions.

As in [4] and “Machine Learning for Statistical Arbitrage II: Feature Engineering and Model Development” on page 2-59, this example uses the current imbalance index I_t and the sign of the forward price movements ΔS_t to build a predictive model for future price movements. The following summarizes these measurements; for more details, see “Machine Learning for Statistical Arbitrage II: Feature Engineering and Model Development” on page 2-59.

Initialize MapReduce for the tall array calculations.

```
mapreducer(0)
```

Convert the tall table into a tall timetable.

```
DT = tall(CombinedDS);
DT.Time = seconds(DT.Time);
DTT = table2timetable(DT);
```

The timestamps are durations, seconds from midnight on June 21, 2012 when the corresponding event occurred. Obtain timestamps that are dates and times of events instead of durations.

```
strt = datetime(201,6,21,0,0,0,34200000);
t = datetime(strt,Format="dd-MMM-yyyy HH:mm:ss.SSSSSSSSSSSSS") + DTT.Time;
```

Imbalance Index I_t

The level 3 imbalance index at time t I_t is the ratio of the weighted averages of the first three levels of ask and bid volumes on either side of the midprice at time t [4]. Symbolically,

$$I_t = \frac{V_{bt} - V_{at}}{V_{bt} + V_{at}},$$

where the level 3 weighted average of bid volumes $V_{bt} = \sum_{j=1}^{3} e^{-\lambda(j-1)} V_{bjt}$ and V_{bjt} is the level j bid volume at time t. V_{at} is similarly defined.

The variable $\lambda > 0$ is a hyperparameter controlling the influence of levels 2 through 3 on the average volume. You choose the value of λ before training or you can tune it by, for example, performing cross-validation. This example sets λ to 0.5.

I_t is in the interval $[-1, 1]$; a positive value indicates a larger bid volume than ask volume and a negative value indicates a larger ask volume than bid volume. A magnitude close to 1 suggests an extreme volume imbalance.

Compute level 3 imbalance index values.

```
lvl = 3;        % Hyperparameter
lambda = 0.5;   % Hyperparameter
weights = exp(-(lambda)*(0:(lvl-1)))';
VAsk = DTT{:,"AskSize" + string(1:lvl)}*weights;
VBid = DTT{:,"BidSize" + string(1:lvl)}*weights;
DTT.I = (VBid-VAsk)./(VBid+VAsk);
```

Midprice S_t

The midprice at time t S_t is the average of the level 1 bid and ask prices, $S_t = (V_{b1t} + V_{a1t})/2$. Price changes are measured with respect to changes in midprices.

Compute the midprices.

```
DTT.Midprices = (DTT.BidPrice1 + DTT.AskPrice1)/2;
```

Focus on Order Executions

Prices are far more likely to change when orders are executed, which correspond to event types 4 and 5 in the message data. Focus the sample on only those events.

```
executionIdx = DTT.Type == 4 | DTT.Type == 5;   % By assumption
I = DTT.I(executionIdx);
```

```
midprices = DTT.Midprices(executionIdx);
t = t(executionIdx);
```

Initiate all calculations on the tall arrays.

```
[I,midprices,t,DTT] = gather(I,midprices,t,DTT);
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 1: Completed in 15 sec
Evaluation completed in 17 sec
```

```
t = t + milliseconds(1:numel(midprices))'; % Make timestamps unique
```

Forward Price Movement ΔS_t

The forward price movement at time t ΔS_t is the difference between the midprice S_t at time t and time $t - \Delta t_S$.

The variable $\Delta t_S > 0$ is a hyperparameter controlling the baseline midprice to compare the current forward price. You can set it to the value informed by theory or practice, or you can tune it.

Following the procedure in “Machine Learning for Statistical Arbitrage II: Feature Engineering and Model Development” on page 2-59 to determine a value for Δt_S, compute the average time for arrivals of order executions.

```
dt = diff(t,1,1);
dt.Format = "s";
dtav = expfit(dt)
```

```
dtav = duration
   0.7214 sec
```

```
dtS = 1;    % Hyperparameter
```

The average arrival time of order executions is 0.72 seconds. The setting $\Delta t_S = 1$ corresponds to an interval of 0.72 seconds, on average, which is close to the figure from the example. Therefore, $\Delta S_t = S_t - S_{t-1}$.

ΔS_t indicates the magnitude and direction of the current price change from the baseline $S_{t-\Delta t_S}$. Although use of the magnitude might inform future price changes, this example discretizes the ΔS_t such that it uses only the direction of the price change.

Compute the forward price movements between execution orders.

```
DS = NaN(size(midprices));
shiftS = midprices((dtS+1):end);
DS(1:(end-dtS)) = sign(shiftS-midprices(1:(end-dtS)));
```

The difference operation introduces Δt_S missing values at the tail of `DS`. Remove the missing values and synchronize all series.

```
DS = DS(1:end-dtS,:);
I = I(dtS+1:end,:);
midprices = midprices(dtS+1:end);
t = t(dtS+1:end,:);
X = [I DS];
```

Response Variable y_t

The goal of the analysis is to build a model that accurately predicts forward price direction $\Delta S_{t+\Delta t_S}$ from historical forward price directions ΔS_j and imbalance index values I_j, $j \le t$.

Prepare the response data. Cast it as a categorical variable.

```
classes = ["loss" "const" "gain"];
y = DS((dtS+1):end);
y = categorical(y,[-1 0 1],classes);
n = numel(y);
numClasses = numel(classes);
```

The predictor data `X` and responses `y` are prepared for many MATLAB® machine learning functions. However, for richer model, the LSTM neural network can associate a time series of predictors with each response.

Display the distribution of y_t.

```
tabulate(y)
```

```
  Value    Count   Percent
   loss     1497      4.61%
  const    29651     91.29%
   gain     1333      4.10%
```

The class distribution is severely imbalanced—a majority of the forward price changes are constant. A naive model that always predicts into the constant class, or a mathematical model trained to predict into that way, has about 91% accuracy. However, such a model does not predict price changes for trading opportunities. Therefore, because the goal is to exploit imbalances to make money, the model needs to be trained to learn how to predict losses and gains well, likely at the expense of misclassifying the constant class.

Prepare Data for LSTM Neural Network

A long short-term memory neural network is a special case of a recurrent neural network (RNN) that associates a time series of features with a response. Its goal is to learn long-term associations between the response and features, while "forgetting" less useful associations. Symbolically,

$$y_t = f(X_t, X_{t-1}, \ldots, X_{t-b}; W),$$

where:

- y_t is the response at time t.
- X_t is the collection of features at time t, which can include lagged response values.
- X_{t-1} is the collection of features at time $t-1$.
- b is the outlook hyperparameter.
- W is the collection of neural network weights and biases.
- f specifies the LSTM architecture.

For more details, see “Long Short-Term Memory Neural Networks” (Deep Learning Toolbox).

Prepare the data for the LSTM neural network by associating each response with the past $b = 100$ forward price movements and imbalance index values (b is arbitrarily chosen in this example). Store each successive feature series in a cell vector.

```
b = 100;    % Hyperpatameter
n = n - b + 1;
XCell = cell(n,1);

for k = 1:n
    tmp = zeros(width(X),b);
    for j = 1:b
        tmp(:,j) = X(j+k-1,:)';
    end
    XCell{k} = tmp;
end
```

Because the first predictor observation contains measurements from times 1 through 100, the response data must start at time 101. In other words, the setting b causes the sample size reduction $n - b$.

Synchronize the series.

```
XCell = XCell(1:(end-1));
y = y(b:(end-1));
midprices = midprices(b-1:end-2);
n = numel(y);
head(table(XCell,y))
```

```
        XCell           y
    ________________    _____

    {2×100 double}    const
    {2×100 double}    const
    {2×100 double}    const
    {2×100 double}    const
    {2×100 double}    const
    {2×100 double}    const
    {2×100 double}    const
    {2×100 double}    const
```

Each response is associated with a 2-by-100 matrix of predictor data. In the first row of the table display, `XCell{1,:}` contains imbalance index values during times `t(1:100)` and `XCell{2,:}` contains forward price movements in the same period. `y(1)` is future forward price movement during `time(101)` (recall $\Delta t_S = 1$). Symbolically, the model is

$$y_{t+1} = f(I_t, I_{t-1}, \ldots, I_{t-100}, \Delta S_t, \Delta S_{t-1}, \ldots, , \Delta S_{t-100}; W).$$

Partition Data

Partition the data into training, validation, and backtesting sets. Because the data are time series, you can use the `tspartition` function, which, among other features, cuts the data, rather than shuffles it before cutting it. Reserve half the data for backtesting, and partition the other half of the data so that 85% is for training the model and 15% is for validation.

```
pBT = 0.5;
tspFull = tspartition(n,Holdout=pBT);
idxTV = training(tspFull);  % Training and validation indices
```

```
idxBT = test(tspFull);      % Backtesting indices
XTV = XCell(idxTV);
yTV = y(idxTV);
XBT = XCell(idxBT);
yBT = y(idxBT);
tBT = t(idxBT);
mpBT = midprices(idxBT);
nBT = sum(idxBT);

pV = 0.15;
nTV = sum(idxTV);
tspTV = tspartition(nTV,Holdout=pV);
idxT = training(tspTV);     % Training indices
idxV = test(tspTV);         % Validation indices
XT = XCell(idxT);
yT = y(idxT);
XV = XCell(idxV);
yV = y(idxV);
```

Configure LSTM Neural Networks

Consider training the following two LSTM neural networks:

- A deep learning model that ignores the severely imbalanced class distribution of the response data.
- A deep learning model that, despite the risk, addresses imbalanced class distribution by applying a weighting scheme.

For both deep learning models, this example uses the following LSTM architecture in this order. For more details, see “Long Short-Term Memory Neural Networks” (Deep Learning Toolbox).

- `sequenceInputLayer` — Sequence input layer for the two feature variables with b = `MinLength` = 100.
- `batchNormalizationLayer` — Batch normalization layer, which normalizes a mini-batch of data across all observations to speed up training and reduce the sensitivity to network initialization.
- `lstmLayer` — Long short-term memory layer with 64 hidden units and configured to return only the last step sequence. The number of hidden layers is a hyperparameter.
- `fullyConnectedLayer` — Fully connected layer, which combines W and inputs, and reduces the number of neurons to 10.
- `leakyReluLayer` — Leaky ReLU layer, which reduces negative outputs of the fully connected layer to 10% of their value.
- `fullyConnectedLayer` — Fully connected layer, which combines W and inputs, and reduces the number of neurons to 3.
- `softmaxLayer` — Softmax layer, which outputs a length 3 vector of class probabilities by applying the softmax function to the outputs of the previous fully connected layer.

The risky model applies the following class weighting scheme to increase the penalty on the loss for predicting into the gain and loss classes:

$$u_j = \frac{n_T}{3c_j},$$

where n_T is the number of training observations and c_j is the number of responses in class j. This weighting scheme forces the classifier to pay more attention to the classes with lower frequency.

Compute the class weights.

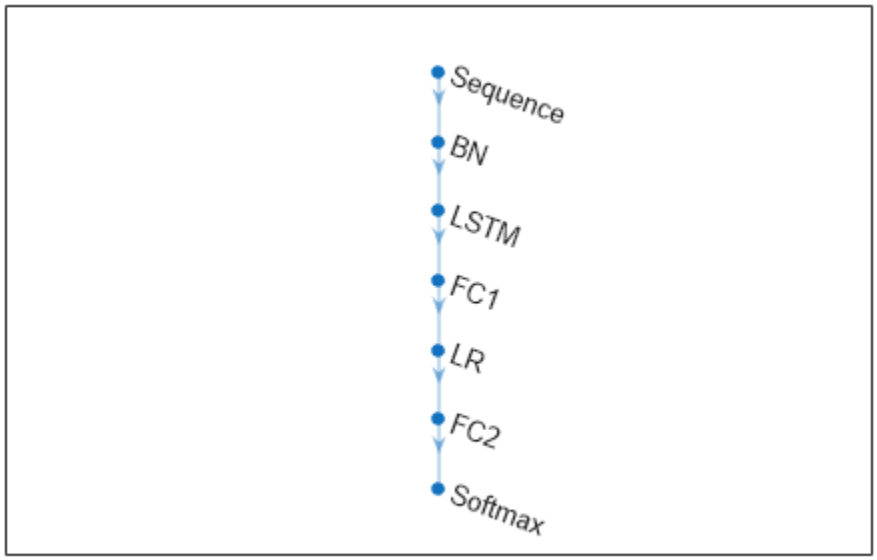
```
nTP = tspTV.TrainSize;
classFreqT = countcats(yT);
classWtsT = nTP./(numClasses*classFreqT);
```

For each LSTM neural network, create an array specifying the architecture. Note that the LSTM architecture itself is a hyperparameter, as are most options taken at their default.

```
% Hyperparameters
numHiddenUnits = 64;
numNeuronsOutFC1 = 10;
thresholdScaleLR = 0.1;

layersLSTM = [
    sequenceInputLayer(2,Name="Sequence",MinLength=b)
    batchNormalizationLayer(Name="BN")
    lstmLayer(numHiddenUnits,Name="LSTM",OutputMode="last")
    fullyConnectedLayer(numNeuronsOutFC1,Name="FC1")
    leakyReluLayer(thresholdScaleLR,Name="LR")
    fullyConnectedLayer(3,Name="FC2")
    softmaxLayer(Name="Softmax")];

lossFcn = @(Y,T) crossentropy(Y,T, ...
    classWtsT, ...
    Reduction = "sum",...
    WeightsFormat="C",...
    ClassificationMode="single-label",...
    NormalizationFactor="none")*numClasses;
```

Create the LSTM architectures and visualize one of them.

```
figure
plot(dlnetwork(layersLSTM))
```



Train and Validate LSTM Neural Network

Set the following training options by using the `trainingOptions` function:

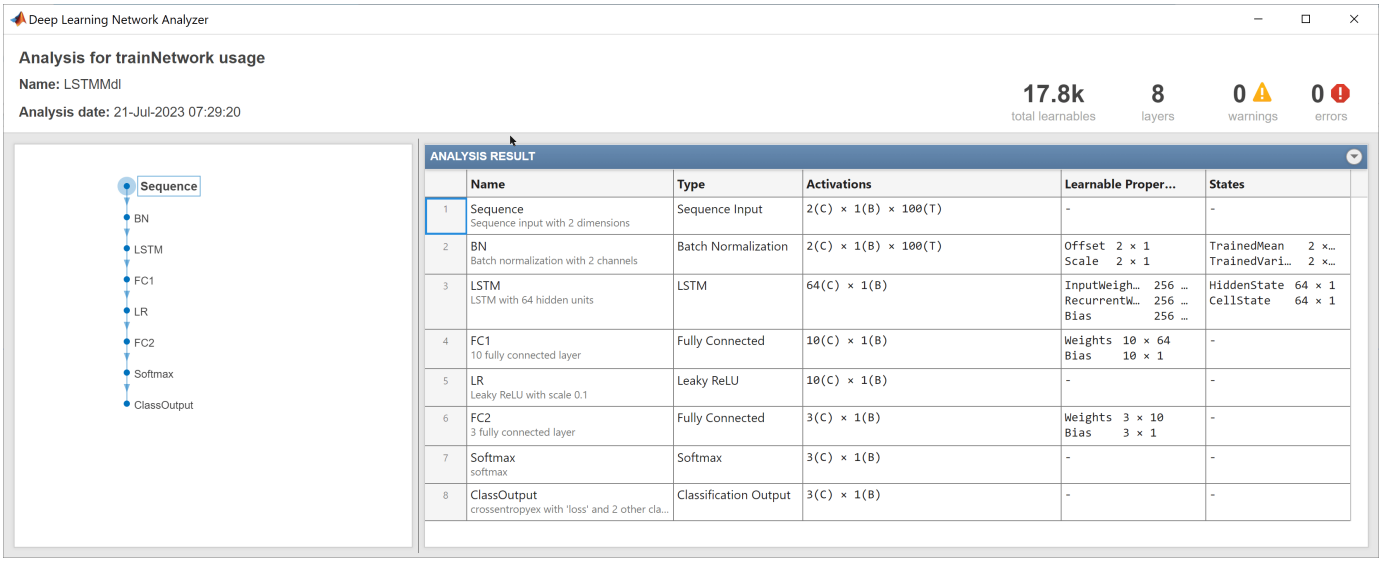
- Use the stochastic gradient descent with momentum solver.
- Plot the training progress.
- Monitor the solver's progress by printing numeric results, in real time, every 100 iterations.
- Set the initial learning rate to `1e-1` and specify a piecewise learning rate schedule. These are hyperparameters.
- Specify the validation data in a cell vector.
- Validate every 100 iterations and set the validation patience to 5.
- Shuffle the data every epoch.

```
VData = {XV,yV};

% Hyperparameters
lr = 1e-4;
lrs = "piecewise";

options = trainingOptions("sgdm", ...
    Plots='training-progress', ...
    InputDataFormats="CTB",...
    Metric="accuracy",...
    Verbose=1,VerboseFrequency=100, ...
    InitialLearnRate=lr,LearnRateSchedule=lrs, ...
    ValidationData=VData,ValidationFrequency=100,ValidationPatience=5, ...
    Shuffle="every-epoch");
```

To train and validate the LSTM neural networks, set `trainLSTM` to `true`. Otherwise, this example loads the pretrained LSTM neural networks `LSTMMdl` and `RiskyLSTMMdl` in the MAT-file

`PretrainedLOBLSTMLvl3.mat`, which is included with the Financial Toolbox documentation. Analyze the trained risky model.

```
trainLSTM = false;
if trainLSTM
    rng(1,"twister")
    LSTMMdl = trainnet(XT,yT,layersLSTM,"crossentropy",options);
    RiskyLSTMMdl = trainnet(XT,yT,layersLSTM,lossFcn,options);
else
    load PretrainedLOBLSTMLvl3
end
analyzeNetwork(RiskyLSTMMdl)
```



Assess Trained LSTM Model Quality

For each trained LSTM model, compute the model accuracy on the validation data and plot a confusion chart by following this procedure:

1. Predict responses and classification scores for the validation data by passing the feature validation data and the trained LSTM model to `classify`.
2. Pass the observed and predicted responses to `confusionchart`. Add column and row summaries to the chart.

```
[predScoresLSTM] = minibatchpredict(LSTMMdl,XV,InputDataFormats="CTB");
```

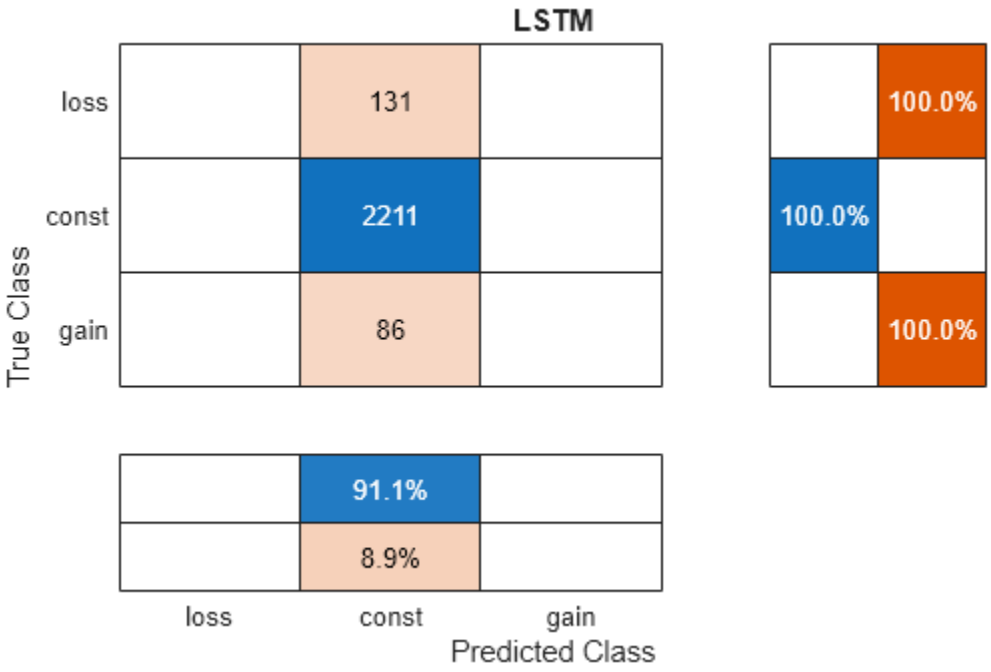
```
Warning: Support for GPU devices with compute capability 7.0 will be removed in a future MATLAB
```

```
predYLSTM = scores2label(predScoresLSTM,classes);
accV = 100*mean(predYLSTM == yV)
```

```
accV =
91.0626
```

```
figure
cm = confusionchart(yV,predYLSTM);
cm.NormalizedValues;
cm.RowSummary = "row-normalized";
```

```
cm.ColumnSummary = "column-normalized";
cm.Title = "LSTM";
```

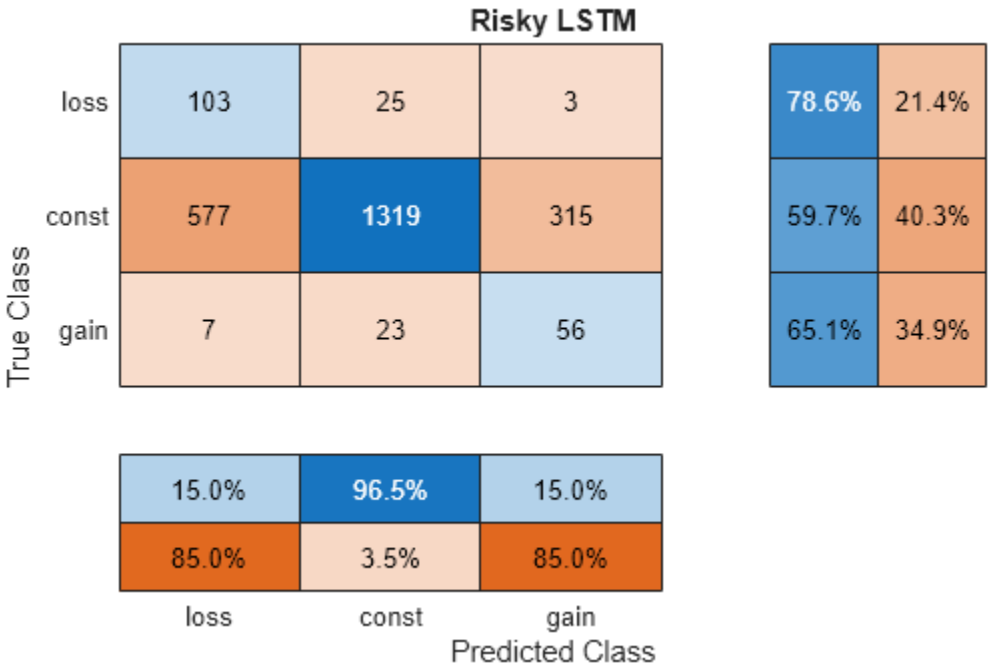


The conservative LSTM neural network predicts into the constant class for all validation observations. The 91% accuracy comes for free because this model acts like the naive model discussed in Engineer Features and Prepare Response Variable on page 2-79, which only ever predicts into the constant class. Such a model does not make money.

```
[predScoresRLSTM] = minibatchpredict(RiskyLSTMMdl,XV,InputDataFormats="CTB");
predYRLSTM = scores2label(predScoresRLSTM,classes);
accRiskyV = 100*mean(predYRLSTM == yV)
```

```
accRiskyV =
60.8731
```

```
figure
cm = confusionchart(yV,predYRLSTM);
cm.NormalizedValues;
cm.RowSummary = "row-normalized";
cm.ColumnSummary = "column-normalized";
cm.Title = "Risky LSTM";
```

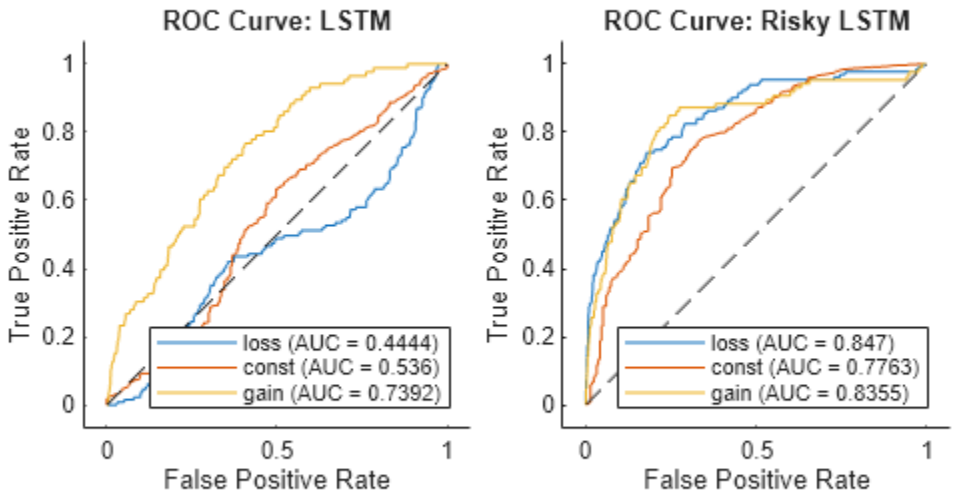


The risky LSTM model accuracy is about 61%. Despite this low value, the confusion chart suggests the following other aspects about the model's performance as it pertains to the trading strategy.

- It correctly predicted about 65% of the 86 true gains, which makes money.
- It correctly predicted about 79% of the 131 true losses, which prevents money loss.
- It incorrectly predicted about 35% of the true gains as losses or constant, which results is a missed opportunity to make money.
- It incorrectly predicted about 2% of the true losses as gains, which results in a money loss.

Plot ROC curves for both deep learning models.

```
rocLSTM = rocmetrics(yV,predScoresLSTM,classes);
rocRLSTM = rocmetrics(yV,predScoresRLSTM,classes);

figure
tiledlayout(1,2,TileSpacing="compact")
nexttile
plot(rocLSTM,ShowModelOperatingPoint=false)
title("ROC Curve: LSTM")
nexttile
plot(rocRLSTM,ShowModelOperatingPoint=false)
title("ROC Curve: Risky LSTM")
```



In general, each curve represents the ability of the model to distinguish observations that are in the corresponding class from the other two classes at various classification decision thresholds. For example, the blue curve corresponds to the ability of the classifiers to correctly distinguish losses from constants or gains. The diagonal dashed line is a hypothetical classifier that assigns observations into classes at random, which is a baseline for comparison. The curve of a good classifier increases very quickly and then levels off, making the area under the curve (AUC) close to 1. In other words, a good classifier has a high true positive rate for all classification decision thresholds, regardless of false positive rate levels produced. Note that the ROC curve does not consider false negative rates.

In this case, the risky LSTM is better at correctly distinguishing each class from the other two than the conservative LSTM; the AUCs of the risky LSTM are larger than the corresponding AUCs of the conservative LSTM.

Backtest Risky LSTM Model

Backtesting a predictive financial model dispatches it, and a trading strategy based on its predictions, on historical data. This action determines how well the model performs, with respect to the strategy. A backtest requires:

- Historical data not used to train or validate the model (see Partition Data on page 2-83)
- The target predictive model (The risky LSTM neural network `RiskyLSTMMdl`)
- The model's predictions (signals) on the historical data
- At least one trading strategy

Classify the forward price movements in the backtest data `XBT` using the risky LSTM neural network.

```
[predScoresRLSTMBT] = minibatchpredict(RiskyLSTMMdl,XBT,InputDataFormats="CTB");
predYRLSTMBT = scores2label(predScoresRLSTMBT,classes);
```

A trading strategy requires a way to predict outcomes and what financial decisions to make based on the outcomes. This example uses the following trading decision logic:

- If the forward price direction is predicted to be a gain, invest all available money.
- If the forward price direction is predicted to be a loss, cash out.
- Otherwise, do nothing.

The `rebalance` function, located in this example in Local Functions on page 2-93, defines these predictive models:

- Maximum Profit Model `MaxProfit`: A baseline model that perfectly predicts the next forward price direction, and therefore it always makes the right trading decisions.
- Risky LSTM Neural Network `RiskyLSTM`: A model that predicts the next forward price direction by using the trained risky LSTM neural network `RiskyLSTMMdl`, with trading decisions based on its predictions.

The inputs of `rebalance` are:

- Required asset weights `w`, which are the proportion of capital to invest across assets
- Required asset prices `p`, which are not used in this example
- Signal data `s`, optional data used to inform trading decisions
- Optional user data structure to print backtest iteration information
- Optional column of signal data to operate on for the strategy
- Optional strategy name

Define the backtest strategies by passing the a function handle to `rebalance` to “Create backtestStrategy” on page 15-233. Set the following properties:

- `LookbackWindow=1`, which indicates that the previous data point is available to the `rebalance` function during backtesting.
- `UserData=user_data`, where `user_data` is a structure array containing the fields `Counter`, set initially to 0, and `NumTestPoints`, set to the backtest sample size.

Store the strategies in a vector.

```
user_data = struct("Counter",0,"NumTestPoints",nBT);
maxMoneyStrat = backtestStrategy("MaxProfit",@(w,p,s,u)rebalance(w,p,s,u,1,"MaxProfit"), ...
    LookbackWindow=1,UserData=user_data);
RiskyLSTMStrat = backtestStrategy("RiskyLSTM",@(w,p,s,u)rebalance(w,p,s,u,2,"RiskyLSTM"), ...
    LookbackWindow=1,UserData=user_data);
strategies = [maxMoneyStrat RiskyLSTMStrat];
```

`strategies` is a vector of `backtestStrategy` objects storing the backtest strategies.

Prepare the backtest engine by supplying the vector of `backtestStrategy` objects to `backtestEngine`.

```
bte = backtestEngine(strategies);
```

`bte` is a `backtestEngine` object specifying parameters for the backtest, including the risk-free rate `RiskFreeRate`, which is 0% by default.

Store the backtest forward price direction data and risky LSTM predictions in a timetable.

```
signalTT = timetable(yBT,predYRLSTMBT,RowTimes=tBT);
```

Store the backtest midprices in a timetable.

```
mpTT = timetable(mpBT,RowTimes=tBT);
```

Run the backtest by passing the backtest engine, midprice data, and signal data to `runBacktest`.

```
bte = runBacktest(bte,mpTT,signalTT);
```

```
Backtesting MaxProfit. Total progress: 5000 of 16190.
Backtesting MaxProfit. Total progress: 10000 of 16190.
Backtesting MaxProfit. Total progress: 15000 of 16190.
Backtesting RiskyLSTM. Total progress: 5000 of 16190.
Backtesting RiskyLSTM. Total progress: 10000 of 16190.
Backtesting RiskyLSTM. Total progress: 15000 of 16190.
```

`bte` is a `backtestEngine` object containing the results of the backtest.

Display a summary of the results by passing the completed backtest engine object to `summary`.

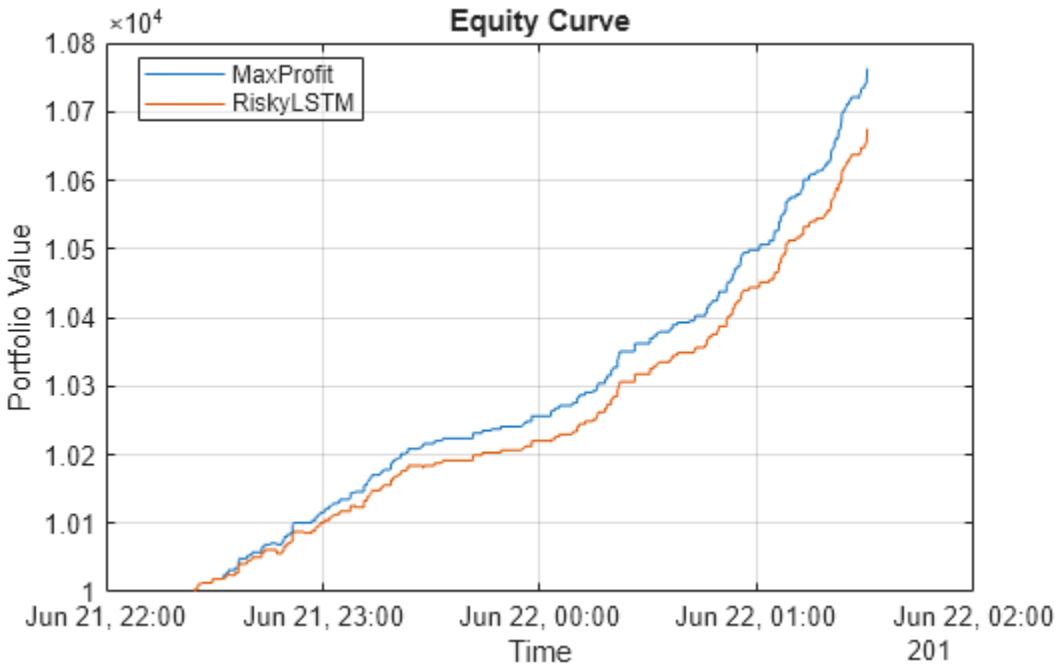
```
summary(bte)
```

```
ans=9×2 table
                       MaxProfit     RiskyLSTM
                       __________    __________

    TotalReturn          0.076328      0.067524
    SharpeRatio           0.15212       0.13715
    Volatility          2.9873e-05    2.9433e-05
    AverageTurnover       0.010624     0.0090802
    MaxTurnover                0.5           0.5
    AverageReturn        4.544e-06    4.0366e-06
    MaxDrawdown         0.00018751    0.00055683
    AverageBuyCost               0             0
    AverageSellCost              0             0
```

The total nonannualized returns of the maximum profit and risky LSTM model are 7.6% and 6.8%, respectively. Therefore, the LSTM neural network performs well over the historical data.

Plot equity curves of the strategies by passing the completed backtest engine object to `equityCurve`.

```
figure
equityCurve(bte)
```



The equity curves indicate both strategies performed well, and are in close correspondence with each other.

Local Function

The `rebalance` function directs the backtest framework to perform the following trades:

- When the model predicts a price gain, invest all available cash.
- When the model predicts a price loss, cash out entirely.
- When the model predicts a constant price, take no actions.

You can use this function only from in this script.

```
function [new_weights,user_data] = rebalance(weights,~,signalData,user_data,col,id)
    new_weights = weights;

    direction = signalData{:,col};
    if direction == "gain"
        new_weights = 1;    % Expect a gain --> Fully invest
    end
    if direction == "loss"
        new_weights = 0;    % Expect loss --> Cash out, with risk-free rate of 0%
    end

    % Print iteration information at command line
    user_data.Counter = user_data.Counter + 1;
    if (mod(user_data.Counter,5000) == 0)
        displayText = "Backtesting " + id + ". Total progress: " + num2str(user_data.Counter) + "
        disp(displayText);
```

```
    end
end
```

References

[1] Kolm, Petter, N., Jeremy Turiel, and Nicholas Westray. "Deep Order Flow Imbalance: Extracting Alpha at Multiple Horizons from the Limit Order Book." *SSRN* (August 2021): https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3900141.

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

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

[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 II: Feature Engineering and Model Development” on page 2-59
- “Machine Learning for Statistical Arbitrage III: Training, Tuning, and Prediction” on page 2-69
- “Backtest Investment Strategies Using Financial Toolbox” on page 4-238
- “Backtest Strategies Using Deep Learning” on page 4-302
- “Backtest Investment Strategies with Trading Signals” on page 4-251
- “Deep Reinforcement Learning for Optimal Trade Execution” on page 4-407

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 3-27
- “Plotting an Efficient Frontier Using portopt” on page 10-22

More About

- “Portfolio Object Workflow” on page 4-18

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 4-18.

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 3-27
- “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 4-18

Portfolio Construction Examples

In this section...

Introduction

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

Note An alternative to using these portfolio optimization functions is to use the Portfolio object (`Portfolio`) for mean-variance portfolio optimization. This object supports gross or net portfolio returns as the return proxy, the variance of portfolio returns as the risk proxy, and a portfolio set that is any combination of the specified constraints to form a portfolio set. For information on the workflow when using Portfolio objects, see “Portfolio Object Workflow” on page 4-18.

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 4-99
- “Obtaining Endpoints of the Efficient Frontier” on page 4-102
- “Obtaining Efficient Portfolios for Target Returns” on page 4-105
- “Obtaining Efficient Portfolios for Target Risks” on page 4-108
- “Efficient Portfolio That Maximizes Sharpe Ratio” on page 4-111
- “Estimate Efficient Frontiers for Portfolio Object” on page 4-122
- “Plotting the Efficient Frontier for a Portfolio Object” on page 4-125

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 3-27
- “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 4-18

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
```

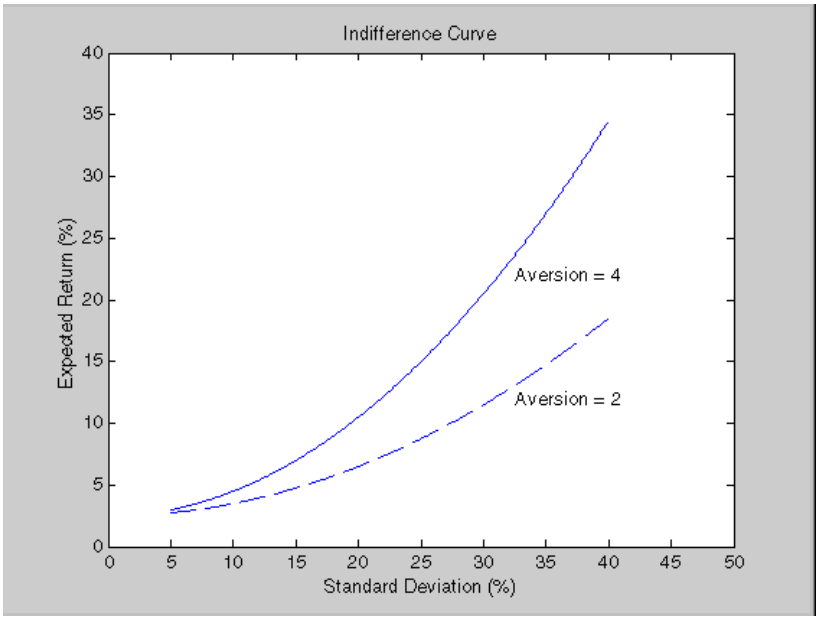
where:

`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 4-18.

Optimal Risky Portfolio

This example shows how to compute 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.

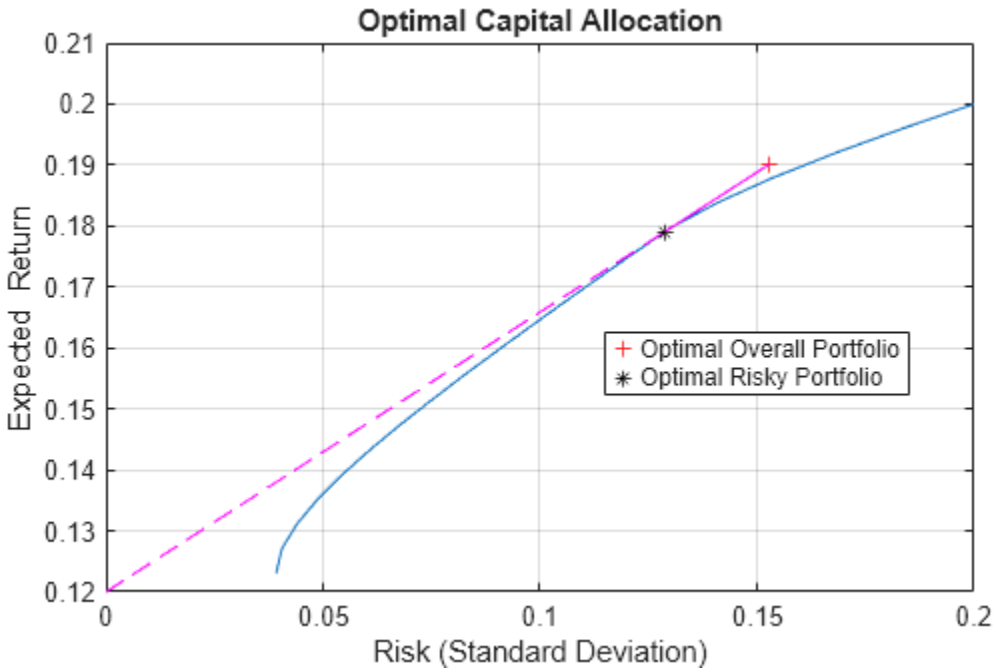
You do this with the function `portalloc`. First generate the efficient frontier data using `portopt`.

```
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];

NumPorts = 20; % Consider 20 different points along the efficient frontier.

[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`.

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

RisklessRate =
0.0800

BorrowRate    =  0.12

BorrowRate =
0.1200

RiskAversion  =  3

RiskAversion =
3

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 = 1×3

    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 3-27
- “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 4-18

portopt Migration to Portfolio Object

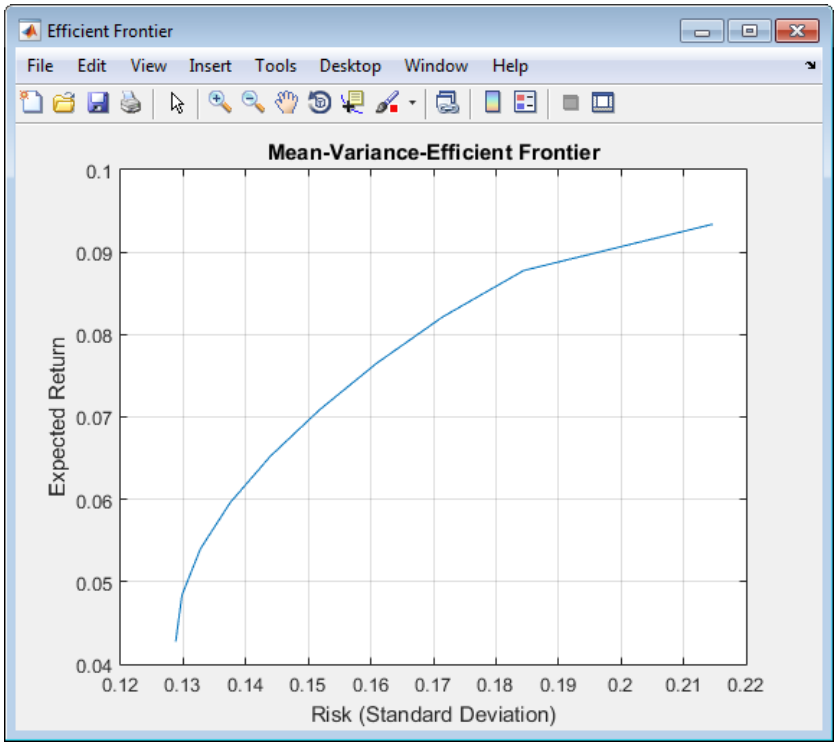
In this section...

Migrate portopt Without Output Arguments

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

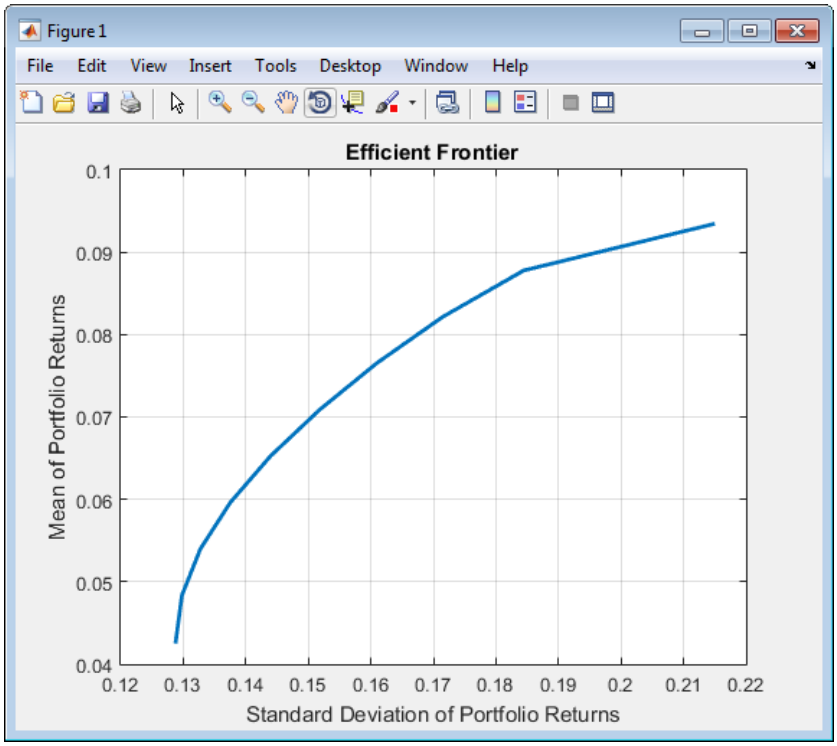
The basic `portopt` functionality is represented as:

```
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 4-18

Constraint Specification Using a Portfolio Object

When constructing the efficient frontier, several constraints are typically considered. You can use linear constraints that define the limits within which the portfolio optimization problem must operate and you can also use group constraints to limit the exposure to certain sectors or groups.

Constraints for Efficient Frontier

This example shows how to compute 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];
```

Use the `Portfolio` object 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
```

```
PortRisk = 8×1

    0.0416
    0.0499
    0.0624
    0.0767
    0.0920
    0.1100
    0.1378
    0.1716
```

```
PortReturn
```

```
PortReturn = 8×1

    0.1279
    0.1361
    0.1442
    0.1524
    0.1605
    0.1687
    0.1768
    0.1850
```

```
PortWts
```

```
PortWts = 3×8

    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

This example shows how to use the `Portfolio` object to specify the minimum and maximum investment in various groups.

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.

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 equation numbers 1, 3 and 5 by –1. Also turn equation number 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 the following:

```
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]
```

```
A = 8×3

    -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]
```

```
b = 8×1

   -0.3000
    0.7500
   -0.1000
    0.5500
   -0.2000
    0.5000
```

```
   -0.5000
    0.5000
```

One approach to solving this portfolio problem is to explicitly use the `setInequality` function with a `Portfolio` object.

```
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 = 8;
AssetBounds = [-0.1, 1.1];

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
```

```
PortRisk = 8×1

    0.0586
    0.0586
    0.0586
    0.0586
    0.0586
    0.0586
    0.0586
    0.0586
```

```
PortReturn
```

```
PortReturn = 8×1

    0.1375
    0.1375
    0.1375
    0.1375
    0.1375
    0.1375
    0.1375
    0.1375
```

```
PortWts
```

```
PortWts = 3×8

    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 example “Specifying Group Constraints” on page 3-24.

Specifying Group Constraints

This example shows how to solve a portfolo problem using a `Portfolio` object with group constraints.

The example “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. The `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 Membership

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

```
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`.

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

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

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

```
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 = 8;
AssetBounds = [-0.1, 1.1];

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
```

```
PortRisk = 8×1

    0.0586
    0.0586
    0.0586
    0.0586
    0.0586
    0.0586
    0.0586
    0.0586
```

```
PortReturn
```

```
PortReturn = 8×1

    0.1375
    0.1375
    0.1375
    0.1375
    0.1375
    0.1375
    0.1375
    0.1375
```

```
PortWts
```

```
PortWts = 3×8

    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 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 4-58
- “Working with 'Simple' Bound Constraints Using Portfolio Object” on page 4-62
- “Working with Budget Constraints Using Portfolio Object” on page 4-65
- “Working with Group Constraints Using Portfolio Object” on page 4-69
- “Working with Group Ratio Constraints Using Portfolio Object” on page 4-72
- “Working with Linear Equality Constraints Using Portfolio Object” on page 4-75
- “Working with Linear Inequality Constraints Using Portfolio Object” on page 4-78
- “Working with Average Turnover Constraints Using Portfolio Object” on page 4-85
- “Working with One-Way Turnover Constraints Using Portfolio Object” on page 4-88
- “Working with Tracking Error Constraints Using Portfolio Object” on page 4-91
- “Asset Allocation Case Study” on page 4-180
- “Portfolio Optimization Examples Using Financial Toolbox” on page 4-161

More About

- Supported Constraints for Portfolio Optimization Using Portfolio Objects on page 4-9
- “Portfolio Object Workflow” on page 4-18
- “Setting Up a Tracking Portfolio” on page 4-40

Active Returns and Tracking Error Efficient Frontier

This example shows how to minimize the variance of the difference in returns with respect to a given target portfolio.

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);
```

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));
```

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)]
```

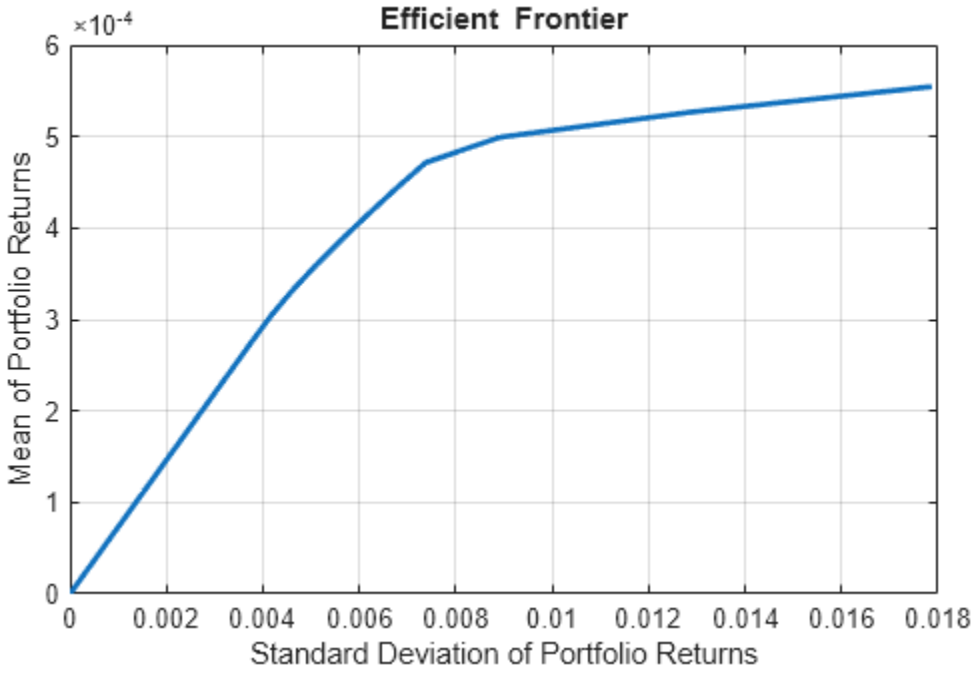
```
ans = 12×2

    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:

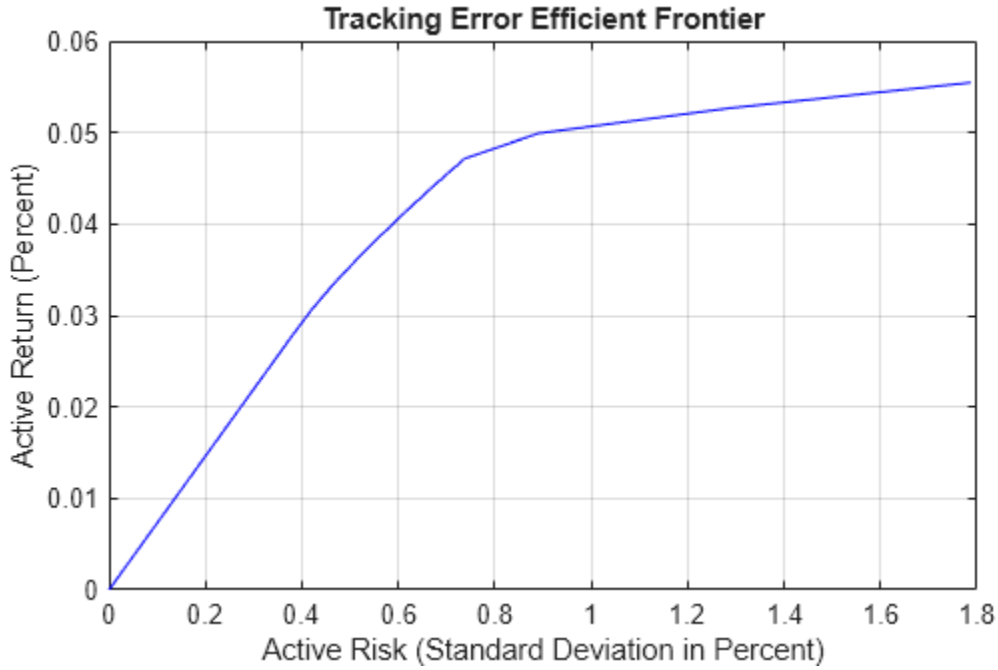
$$b_{active} = b_{absolute} - A \times Index.$$

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)
```

```
ActiveWeights = 5×21

   -0.0000   -0.0038   -0.0075   -0.0113   -0.0151   -0.0188   -0.0226   -0.0264   -0.0302   -0.0
    0.0000   -0.0026   -0.0052   -0.0078   -0.0104   -0.0130   -0.0156   -0.0182   -0.0209   -0.0
    0.0000    0.0093    0.0186    0.0279    0.0372    0.0465    0.0559    0.0652    0.0745    0.0
   -0.0000   -0.0192   -0.0384   -0.0576   -0.0768   -0.0960   -0.1152   -0.1344   -0.1536   -0.1
    0.0000    0.0163    0.0325    0.0488    0.0651    0.0814    0.0976    0.1139    0.1302    0.1
```

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

```
AbsoluteWeights = 5×21

    0.2000    0.1962    0.1925    0.1887    0.1849    0.1812    0.1774    0.1736    0.1698    0.1
    0.2000    0.1974    0.1948    0.1922    0.1896    0.1870    0.1844    0.1818    0.1791    0.1
    0.2000    0.2093    0.2186    0.2279    0.2372    0.2465    0.2559    0.2652    0.2745    0.2
    0.2000    0.1808    0.1616    0.1424    0.1232    0.1040    0.0848    0.0656    0.0464    0.0
    0.2000    0.2163    0.2325    0.2488    0.2651    0.2814    0.2976    0.3139    0.3302    0.3
```

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 4-18

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

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

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

and (sample) covariance of asset returns

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

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

Gross Portfolio Returns

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

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

where:

r_0 is the risk-free rate (scalar).

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

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

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

Net Portfolio Returns

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

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

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 $\sigma: X \to R$ 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 $(1 - \alpha)$. Given $VaR_\alpha(x)$ for a portfolio *x*, the conditional value-at-risk of the portfolio is the expected loss of portfolio returns above the value-at-risk return.

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

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

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

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 Portfolio Object” on page 4-25
- “Working with Portfolio Constraints Using Defaults” on page 4-58
- “Asset Allocation Case Study” on page 4-180
- “Portfolio Optimization Examples Using Financial Toolbox” on page 4-161

More About

- `Portfolio`
- “Portfolio Object Workflow” on page 4-18
- Supported Constraints for Portfolio Optimization Using Portfolio Objects on page 4-9
- “Default Portfolio Problem” on page 4-17
- “Role of Convexity in Portfolio Problems” on page 4-157

External Websites

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

Supported Constraints for Portfolio 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'` Bound constraints
- Budget constraints
- Conditional 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

$$A_I x \le b_I$$

where:

x is the portfolio (n vector).

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

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

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

`Portfolio`, `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 4-17) 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 4-17) 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 4-5 and “Working with a Riskless Asset” on page 4-52).

Conditional Budget Constraints

Conditional budget constraints is a logical constraint that bounds the maximum amount of the portfolio that can be invested in assets exceeding a given threshold. This constraint supports the “Undertakings for Collective Investment in Transferable Securities” on page 15-1452 (UCITS) directive. Use `setConditionalBudget` to set conditional budget constraints. To mathematically formulate this type of constraints, a binary variable w_i is needed. If $w_i = 0$ the asset is below the specified threshold, otherwise the asset is above that threshold which is represented as

$$w_i = 0 \rightarrow x_i \leq \alpha_i$$

$$\sum_i x_i w_i \leq \gamma$$

where:

x is an n vector representing the portfolio.

α is a scalar or n representing the threshold above which assets are accounted for in the conditional budget (scalar or n vector).

γ is a scalar representing the maximum amount of the portfolio that can be invested in assets exceeding α.

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 4-85) 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

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

where:

x is the portfolio (n vector).

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

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

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

The default is to ignore this constraint.

See Also

`Portfolio` | `PortfolioCVaR` | `PortfolioMAD`

Related Examples

- “Creating the Portfolio Object” on page 4-25
- “Working with Portfolio Constraints Using Defaults” on page 4-58
- “Asset Allocation Case Study” on page 4-180
- “Portfolio Optimization Examples Using Financial Toolbox” on page 4-161
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-190
- “Black-Litterman Portfolio Optimization Using Financial Toolbox” on page 4-222
- “Portfolio Optimization Using Factor Models” on page 4-231

More About

- `Portfolio`
- “Portfolio Object Workflow” on page 4-18
- “Default Portfolio Problem” 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)

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 4-25
- “Working with Portfolio Constraints Using Defaults” on page 4-58
- “Asset Allocation Case Study” on page 4-180
- “Portfolio Optimization Examples Using Financial Toolbox” on page 4-161
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-190
- “Black-Litterman Portfolio Optimization Using Financial Toolbox” on page 4-222
- “Portfolio Optimization Using Factor Models” on page 4-231
- “Portfolio Optimization Using Social Performance Measure” on page 4-264
- “Diversify Portfolios Using Custom Objective” on page 4-331

More About

- `Portfolio`
- Supported Constraints for Portfolio Optimization Using Portfolio Objects on page 4-9
- “Portfolio Object Workflow” on page 4-18

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 4-25.

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 4-42.

3. Specify the Portfolio Constraints.

   Define the constraints for portfolio assets such as linear equality and inequality, bound, budget, conditional 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 4-58 and “Working with 'Conditional' BoundType, MinNumAssets, and MaxNumAssets Constraints Using Portfolio Objects” on page 4-81.

4. Validate the Portfolio.

   Identify errors for the portfolio specification. For more information, see “Validate the Portfolio Problem for Portfolio Object” on page 4-94.

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 4-98 and “Estimate Efficient Frontiers for Portfolio Object” on page 4-122.

   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 4-119.

6. Postprocess the results.

   Use the efficient portfolios and efficient frontiers results to set up trades. For more information, see “Postprocessing Results to Set Up Tradable Portfolios” on page 4-130.

For an example of this workflow, see “Asset Allocation Case Study” on page 4-180 and “Portfolio Optimization Examples Using Financial Toolbox” on page 4-161.

See Also

Related Examples

- “Asset Allocation Case Study” on page 4-180
- “Portfolio Optimization Examples Using Financial Toolbox” on page 4-161
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-190
- “Black-Litterman Portfolio Optimization Using Financial Toolbox” on page 4-222

- “Portfolio Optimization Using Factor Models” on page 4-231
- “Portfolio Optimization Using Social Performance Measure” on page 4-264
- “Diversify Portfolios Using Custom Objective” on page 4-331

More About

- “Portfolio Optimization Theory” on page 4-4
- “Choosing and Controlling the Solver for Mean-Variance Portfolio Optimization” on page 4-114

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 4-25.

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
    ConditionalBudgetThreshold
    ConditionalUpperBudget
    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 4-25
- “Working with Portfolio Constraints Using Defaults” on page 4-58
- “Asset Allocation Case Study” on page 4-180
- “Portfolio Optimization Examples Using Financial Toolbox” on page 4-161
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-190
- “Black-Litterman Portfolio Optimization Using Financial Toolbox” on page 4-222

- “Portfolio Optimization Using Factor Models” on page 4-231
- “Portfolio Optimization Using Social Performance Measure” on page 4-264
- “Diversify Portfolios Using Custom Objective” on page 4-331

More About

- “Portfolio Optimization Theory” on page 4-4
- “Portfolio Object Workflow” on page 4-18

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 4-18.

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 4-30). 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: []
    ConditionalBudgetThreshold: []
```

```
        ConditionalUpperBudget: []
                     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`:

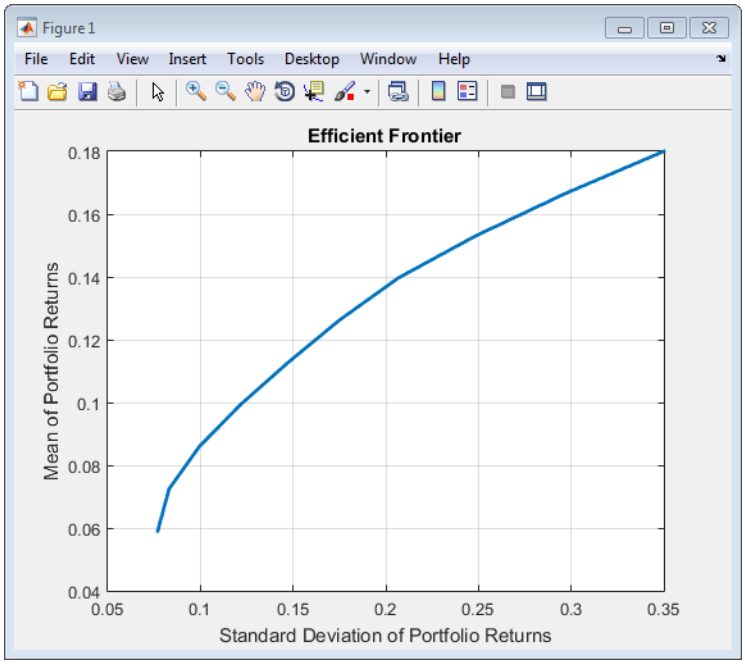
```
m = [ 0.05; 0.1; 0.12; 0.18 ];
C = [ 0.0064 0.00408 0.00192 0;
    0.00408 0.0289 0.0204 0.0119;
    0.00192 0.0204 0.0576 0.0336;
    0 0.0119 0.0336 0.1225 ];

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: []
    ConditionalBudgetThreshold: []
```

```
ConditionalUpperBudget: []
             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`.

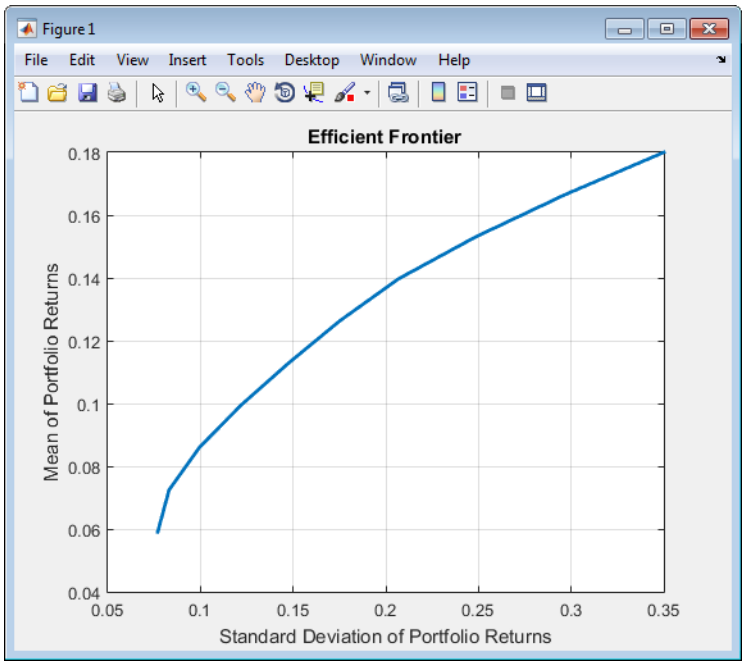
```
p.plotFrontier
```



Using the Portfolio Function with a Sequence of Steps

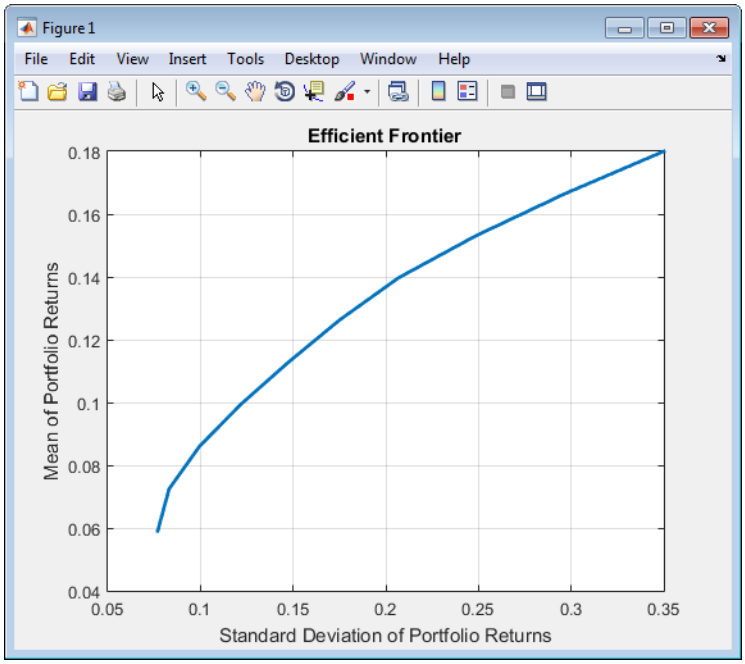
An alternative way to accomplish the same task of setting up a “standard” portfolio optimization problem, given a mean and covariance of asset returns in the variables `m` and `C` (which also illustrates that argument names are not case-sensitive):

```
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 4-33
- “Working with Portfolio Constraints Using Defaults” on page 4-58
- “Asset Allocation Case Study” on page 4-180
- “Portfolio Optimization Examples Using Financial Toolbox” on page 4-161
- “Portfolio Optimization with Semicontinuous and Cardinality Constraints” on page 4-190
- “Black-Litterman Portfolio Optimization Using Financial Toolbox” on page 4-222
- “Portfolio Optimization Using Factor Models” on page 4-231
- “Diversify Portfolios Using Custom Objective” on page 4-331
- “Portfolio Optimization Using Social Performance Measure” on page 4-264

More About

- “Portfolio Object” on page 4-20
- “Portfolio Optimization Theory” on page 4-4
- “Portfolio Object Workflow” on page 4-18

External Websites

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