Financial Toolbox

For Use with MATLAB®

- Computation
- Visualization
- Programming

User's Guide

Version 2

The MathWorks

How to Contact The MathWorks:

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`support@mathworks.com`	Technical support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information
508-647-7000	Phone
508-647-7001	Fax
The MathWorks, Inc. 3 Apple Hill Drive Natick, MA 01760-2098	Mail

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

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)

Contents

1 Getting Started

2 Tutorial

3 Portfolio Analysis

4 Solving Sample Problems

5 Function Reference

A Bibliography

Glossary

Index

1

Getting Started

What Is the Financial Toolbox?

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

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

With MATLAB and the Financial Toolbox, you can

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

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

- “Using Matrix Functions for Finance” on page 1-4

  Reviews “Key Definitions” on page 1-4 and some matrix algebra fundamentals, such as “Referencing Matrix Elements” on page 1-4 and “Transposing Matrices” on page 1-6.

- “Matrix Algebra Refresher” on page 1-7

  Provides a brief refresher on using matrix functions in financial analysis and engineering

- “Function Input/Output Arguments” on page 1-18

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

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

Using Matrix Functions for Finance

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

Key Definitions

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

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

Vector. A matrix with only one row or column. Described as a “1-by-n” or “m-by-1” matrix. The description is always “row-by-column.” Here is a 1-by-4 vector of cash flows in MATLAB notation.

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

Scalar. A 1-by-1 matrix; i.e., a single number.

Referencing Matrix Elements

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

```
Bonds(1,2)

ans =

         0.06
```

```
Cash(3)

ans =

      5280.00
```

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

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

adds another row to the matrix and creates

```
Bonds =

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

Likewise,

```
Prices = [987.50
          475.00
          995.00]

Bonds = [Prices, Bonds]
```

adds another column and creates

```
Bonds =

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

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

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

BondItems =

    500.00   0.055   4
```

Transposing Matrices

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

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

produces

```
Cash =

        1500
        4470
        5280
       -1299
```

Matrix Algebra Refresher

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

- "Adding and Subtracting Matrices" on page 1-7
- "Multiplying Matrices" on page 1-8
- "Dividing Matrices" on page 1-13
- "Solving Simultaneous Linear Equations" on page 1-13
- "Operating Element-by-Element" on page 1-17

These explanations should help refresh your skills.

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

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

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

Adding and Subtracting Matrices

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

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

```
Portfolios_RS = [175   125
                 200   200
                 100   500];

NewPortfolios = Portfolios_PQ + Portfolios_RS

NewPortfolios =

        275.00        325.00
        700.00        600.00
        400.00        650.00
```

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

```
SmallerPortf = NewPortfolios-10

SmallerPortf =
        265.00        315.00
        690.00        590.00
        390.00        640.00
```

Multiplying Matrices

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

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

Multiplying Vectors

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

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

The portfolio contains these numbers of shares of each stock.

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

To find the value of the portfolio, simply multiply the vectors

```
PortfValue = ClosePrices * NumShares
```

which yields

```
PortfValue =

           35412.50
```

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

To illustrate order dependence, switch the order of the vectors

```
Values = NumShares * ClosePrices

Values =

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

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

Computing Dot Products of Vectors

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

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

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

then the dot product

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

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

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

Value =

      35412.50

Value = sum(ClosePrices .* NumShares')

Value =

      35412.50
```

As expected, the value in these cases is exactly the same as the `PortfValue` computed previously.

Multiplying Vectors and Matrices

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

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

```
PortQuan = [100
            500
            300];
```

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

```
WeekPortValue = WeekClosePr * PortQuan

WeekPortValue =

      35412.50
      35587.50
      35475.00
      35550.00
      35512.50
```

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

Multiplying Two Matrices

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

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

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

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

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

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

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

```
PortfolioValues = WeekClosePr * Portfolios

PortfolioValues =

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

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

Multiplying a Matrix by a Scalar

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

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

DoublePort = Portfolios * 2

DoublePort =
          200.00        400.00
         1000.00        800.00
          600.00        300.00
```

Dividing Matrices

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

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

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

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

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

Solving Simultaneous Linear Equations

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

A linear equation is any equation of the form

$$a_1x + a_2y = b$$

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

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

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

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

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

where $A{*}X = B$.

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

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

B = [13
    -18];

X = A \ B
```

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

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

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

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

by representing them as matrices

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

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

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

As word equations,

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

As MATLAB matrices,

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

Units =[10
        20];

Receipts = Cash * Units

Receipts =

       2400.00
       1600.00
```

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

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

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

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

Receipts = [7000
            5000];

Units = Cash \ Receipts
Units =

         43.75
         37.50
```

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

Operating Element-by-Element

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

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

Yields = Dividends ./ Prices

Yields =

    0.0741    0.0225    0.0597    0.0744
```

Function Input/Output Arguments

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

Input Arguments

Matrix Input

Because MATLAB can process vectors and matrices easily, most functions in the Financial Toolbox allow vector or matrix input arguments, rather than just single (scalar) values.

For example, the `irr` function computes the internal rate of return of a cash flow stream. It accepts a vector of cash flows and returns a scalar-valued internal rate of return. However, it also accepts a matrix of cash flow streams, a column in the matrix representing a different cash flow stream. In this case, `irr` returns a vector of internal rates of return, each entry in the vector corresponding to a column of the input matrix. Many other toolbox functions work similarly.

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

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

which MATLAB displays as

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

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

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

The internal rate of return of this investment is

```
Rate =

    0.1201
```

or 12.01%.

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

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

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

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

```
Rate =

    0.1201    0.1201    0.1201
```

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

Matrices of String Input

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

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

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

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

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

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

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

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

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

```
Item =

SZc14-Sep-1999
```

Function Output Arguments

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

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

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

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

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

Some functions that return vectors accept only scalars as arguments. Why could such functions not accept vectors as arguments and return matrices, where each column in the output matrix corresponds to an entry in the input vector? The answer is that the output vectors can be variable length and thus will not fit in a matrix without some convention to indicate that the shorter columns are missing data.

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

`amortize`	Amortization
`depfixdb`	Fixed declining-balance depreciation
`depgendb`	General declining-balance depreciation
`depsoyd`	Sum of years' digits depreciation

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

Interest Rate Arguments

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

2

Tutorial

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

- Handling and converting dates

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

- Formatting currency

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

- Charting financial data

  Charting functions produce a variety of financial charts including Bollinger bands, high-low-close charts, candlestick plots, point and figure plots, and moving-average plots. The Financial Time Series Toolbox provides additional charting functions. See the *Financial Time Series Toolbox User's Guide* for a description of these functions.

- Analyzing and computing cash flows

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

- Pricing and computing yields for fixed-income securities; analyzing the term structure of interest rates

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

- Pricing and analyzing equity derivatives

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

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

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

- Analyzing portfolios

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

The toolbox also contains sets of functions for modeling volatility in time series.

- **Generalized Autoregressive Conditional Heteroskedasticity (GARCH)** functions model the volatility of univariate economic time series. (The GARCH Toolbox provides a more comprehensive and integrated computing environment. For more information see the GARCH Toolbox documentation or the financial products Web page at `http://www.mathworks.com/products/finprod`.)

Handling and Converting Dates

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

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

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

Date Formats

You most often work with date strings (14-Sep-1999) when dealing with dates. The Financial Toolbox works internally with *serial date numbers* (e.g., 730377). A serial date number represents a calendar date as the number of days that has passed since a fixed base date. In MATLAB, serial date number 1 is January 1, 0000 A.D. MATLAB also uses serial time to represent fractions of days beginning at midnight; for example, 6 p.m. equals 0.75 serial days. So 6:00 pm on 14-Sep-1999, in MATLAB, is date number `730377.75`.

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

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

Date Conversions

The toolbox provides functions that convert between date formats.

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

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

Input Conversions

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

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

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

```
'17-May-1999'
'17-May-99'
```

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

and both of these represent the same time.

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

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

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

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

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

Then call the function with the variables

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

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

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

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

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

The decision to represent dates as either date strings or serial date numbers is often a matter of convenience. For example, when formatting data for visual display or for debugging date-handling code, it is often much easier to view dates as date strings because serial date numbers are difficult to interpret. Alternatively, serial date numbers are just another type of numeric data, and can be placed in a matrix along with any other numeric data for convenient manipulation.

Remember that if you create a vector of input date strings, use a column vector and be sure all strings are the same length. Fill with spaces or zeros. See "Matrices of String Input" on page 1-19.

Output Conversions

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

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

Format	Description
`15:45:17`	hour:minute:second
`03:45:17 PM`	hour:minute:second AM or PM
`15:45`	hour:minute
`03:45 PM`	hour:minute AM or PM
`Q1-99`	calendar quarter-year
`Q1`	calendar quarter

Current Date and Time

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

```
today

ans =
      730693

now

ans =

      730693.48
```

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

```
date

ans =

26-Jul-2000
```

Determining Dates

The toolbox provides many functions for determining specific dates, including functions which account for holidays and other nontrading days.

For example, you schedule an accounting procedure for the last Friday of every month. The `lweekdate` function returns those dates for 2000; the 6 specifies Friday.

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

Fridays = datestr(Fridates)

Fridays =

28-Jan-2000
25-Feb-2000
31-Mar-2000
28-Apr-2000
26-May-2000
30-Jun-2000
28-Jul-2000
25-Aug-2000
29-Sep-2000
27-Oct-2000
24-Nov-2000
29-Dec-2000
```

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

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

MLKDays = datestr(MLKDates)

MLKDays =

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

Accounting for holidays and other nontrading days is important when examining financial dates. The toolbox provides the `holidays` function, which contains holidays and special nontrading days for the New York Stock Exchange between 1950 and 2030, inclusive. You can edit the `holidays.m` file to customize it with your own holidays and nontrading days. In this example, use it to determine the standard holidays in the last half of 2000.

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

LHHDays = datestr(LHHDates)

LHHDays =

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

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

```
LHNextDates = busdate(LHHDates);

LHNextDays = datestr(LHNextDates)

LHNextDays =

05-Jul-2000
05-Sep-2000
24-Nov-2000
26-Dec-2000
```

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

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

PayDays = datestr(PayDates)

PayDays =

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

Formatting Currency

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

`cur2frac`	Converts decimal currency values to fractional values
`cur2str`	Converts a value to Financial Toolbox bank format
`frac2cur`	Converts fractional currency values to decimal values

These examples show their use.

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

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

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

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

Charting Financial Data

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

`bolling`	Bollinger band chart
`candle`	Candlestick chart
`pointfig`	Point and figure chart
`highlow`	High, low, open, close chart
`movavg`	Leading and lagging moving averages chart

These functions work with standard MATLAB functions that draw axes, control appearance, and add labels and titles. For users having additional charting requirements, the Financial Time Series Toolbox provides a more comprehensive set of charting functions.

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

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

High-Low-Close Chart Example

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

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

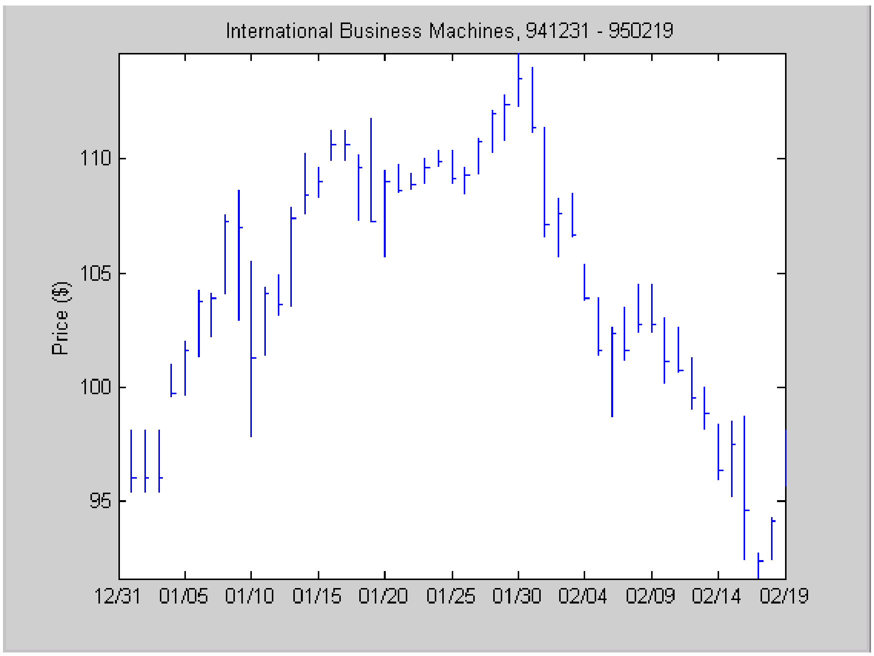
Open a figure window for the chart. Use the Financial Toolbox `highlow` function to plot high, low, and close prices for the last 50 trading days in the data file.

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

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

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

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



Bollinger Chart Example

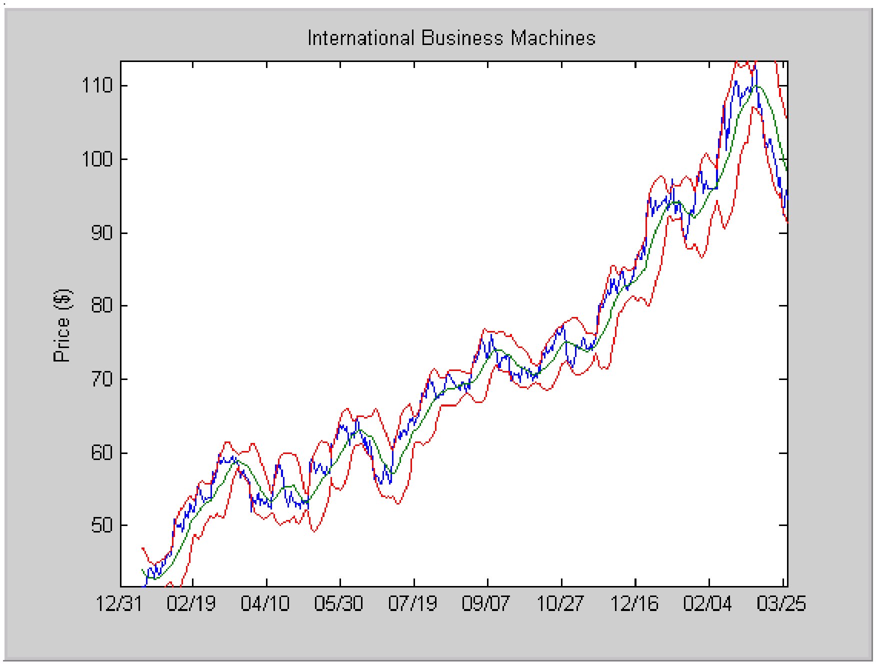
Next the Financial Toolbox `bolling` function produces a Bollinger band chart using all the closing prices in the same IBM stock price matrix. A Bollinger band chart plots actual data along with three other bands of data. The upper band is two standard deviations above a moving average; the lower band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

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

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

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

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



For help using MATLAB plotting functions, see “Creating Plots” in the MATLAB documentation. See the MATLAB documentation for details on the `axis`, `title`, `xlabel`, and `ylabel` functions.

Analyzing and Computing Cash Flows

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

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

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

Interest Rates/Rates of Return

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

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

which gives a rate of return of 11.72%.

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

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

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

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

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

The answer is 9.38%.

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

Present or Future Values

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

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

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

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

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

Depreciation

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

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

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

which returns

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

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

Annuities

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

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

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

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

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

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

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

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

This function returns vectors containing the amount of principal paid,

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

the amount of interest paid,

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

the remaining balance for each period of the loan,

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

and a scalar for the monthly payment.

```
Payment = 437.26
```

Pricing and Computing Yields for Fixed-Income Securities

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

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

The online Function Reference identifies SIA-compliant functions. These functions have been thoroughly tested against the benchmarks found in Jan Mayle's book listed in Appendix A, "Bibliography."

Terminology

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

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

The *first coupon date* and *last coupon date* are the dates when the first and last coupons are paid, respectively. Although bonds typically pay periodic annual or semiannual coupons, the length of the first and last coupon periods may differ from the standard coupon period. The toolbox includes price and yield functions that handle these odd first and/or last periods.

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

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

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

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

Table 2-1: Period of a Bond

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

The *basis* of a bond refers to the basis or day-count convention for a bond. Basis is normally expressed as a fraction in which the numerator determines the number of days between two dates, and the denominator determines the number of days in the year. For example, the numerator of *actual/actual* means that when determining the number of days between two dates, count the actual number of days; the denominator means that you use the actual number of days in the given year in any calculations (either 365 or 366 days depending on whether or not the given year is a leap year).

Table 2-2: Basis of a Bond

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

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

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

Table 2-3: End-of-Month Rule

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

SIA Framework

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

Table 2-4: SIA Common Input Arguments

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

Table 2-4: SIA Common Input Arguments (Continued)

Input	Meaning
`EndMonthRule`	End-of-month payment rule
`IssueDate`	Bond issue date
`FirstCouponDate`	First coupon payment date
`LastCouponDate`	Last coupon payment date

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

SIA Default Parameter Values

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

The complete calling syntax with the full input argument list is

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

while the minimal calling syntax requires only settlement and maturity dates

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

Single Bond Example

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

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

```
IssueDate       = NaN
FirstCouponDate = NaN
LastCouponDate  = NaN
```

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

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

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

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

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

Bond Portfolio Example

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

Now suppose you have a portfolio of two bonds.

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

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

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

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

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

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

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

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

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

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

SIA Coupon Date Calculations

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

The first step in finding the coupon dates associated with a bond is to determine the reference, or synchronization date (the *sync date*). Within the SIA framework, the order of precedence for determining the sync date is (1) the first coupon date, (2) the last coupon date, and finally (3) the maturity date.

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

SIA Semiannual Yield Conventions

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

Pricing Functions

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

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

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

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

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

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

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

Yield Functions

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

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

```
IssueDate       = '01-Jan-2000';
FirstCouponDate = '15-Jan-2000';
LastCouponDate  = '15-Apr-2000';
```

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

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

Calling the function

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

returns

`Yield = 0.0659` (6.60%).

Fixed-Income Sensitivities

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

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

```
CouponRate = 0.05;
```

```
Yield = 0.045;

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

The durations are

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

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

Term Structure of Interest Rates

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

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

First, capture Treasury bill quotes in their reported format

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

and then capture Treasury bond quotes in their reported format

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

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

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

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

```
TBTBond = tbl2bond(TBill)

TBTBond =
         0     730846     99.26      99.27     0.05
```

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

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

```
TBondsAll = [TBTBond; TBond]

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

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

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

With this market data, you are now ready to use one of the toolbox bootstrapping functions to derive an implied zero curve. Bootstrapping is a process whereby you begin with known data points and solve for unknown data points using an underlying arbitrage theory. Every coupon bond can be valued as a package of zero-coupon bonds which mimic its cash flow and risk characteristics. By mapping yields-to-maturity for each theoretical zero-coupon bond, to the dates spanning the investment horizon, you can create a theoretical zero-rate curve.

The toolbox provides two bootstrapping functions. `zbtprice` derives a zero curve from bond data and *prices*, and `zbtyield` derives a zero curve from bond data and *yields*. Using `zbtprice`

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

ZeroRates =

     0.05
     0.06

CurveDates =

      730846
      731160
```

CurveDates gives the investment horizon.

```
datestr(CurveDates)

ans =

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

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

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

Pricing and Analyzing Equity Derivatives

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

Sensitivity Measures

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

Delta

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

Gamma

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

Lambda

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

Rho

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

Theta

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

Vega

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

Implied Volatility

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

Analysis Models

Toolbox functions for analyzing equity derivatives use the Black-Scholes model for European options and the binomial model for American options. The Black-Scholes model makes several assumptions about the underlying securities and their behavior. The binomial model, on the other hand, makes far fewer assumptions about the processes underlying an option. For further explanation, see John Hull's book listed in Appendix A, "Bibliography."

Black-Scholes Model

Using the Black-Scholes model entails several assumptions:

- The prices of the underlying asset follow an Ito process. (See Hull, page 222.)
- The option can be exercised only on its expiration date (European option).
- Short selling is permitted.
- There are no transaction costs.
- All securities are divisible.

- There is no riskless arbitrage.
- Trading is a continuous process.
- The risk-free interest rate is constant and remains the same for all maturities.

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

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

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

yields:

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

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

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

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

Binomial Model

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

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

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

returns the tree of prices of the underlying asset

```
StockPrice =

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

and the tree of option values.

```
OptionPrice =

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

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

3

Portfolio Analysis

Analyzing Portfolios

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

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

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

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

Portfolio Optimization Functions

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

Capital Allocation	
portalloc	Computes the optimal risky portfolio on the efficient frontier, based on the risk-free rate, the borrowing rate, and the investor's degree of risk aversion. Also generates the capital allocation line, which provides the optimal allocation of funds between the risky portfolio and the risk-free asset.

Efficient Frontier Computation	
frontcon	Computes portfolios along the efficient frontier for a given group of assets. The computation is based on sets of constraints representing the maximum and minimum weights for each asset, and the maximum and minimum total weight for specified groups of assets.
portopt	Computes portfolios along the efficient frontier for a given group of assets. The computation is based on a set of user-specified linear constraints. Typically, these constraints are generated using the constraint specification functions described below.

Constraint Specification	
portcons	Generates the portfolio constraints matrix for a portfolio of asset investments using linear inequalities. The inequalities are of the type `A*Wts' <= b`, where `Wts` is a row vector of weights. The capabilities of `portcons` are also provided individually by the following functions.

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

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

Portfolio Construction Examples

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

Efficient Frontier Example

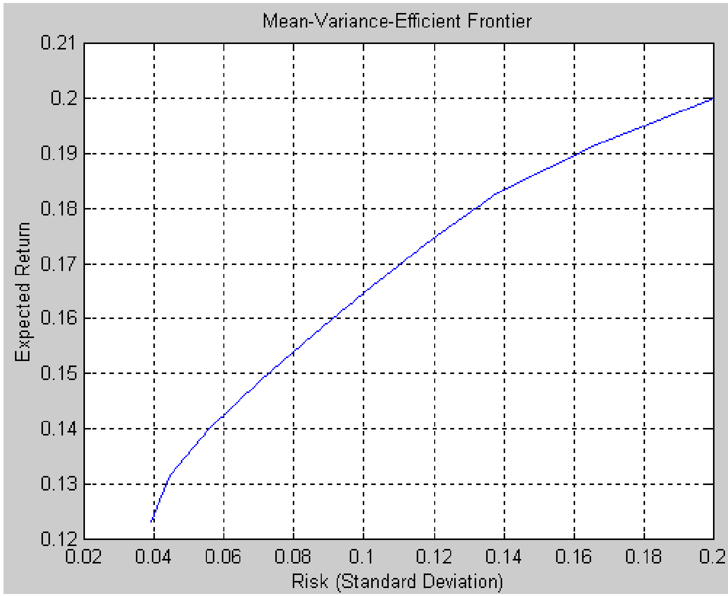
This example computes the efficient frontier of portfolios consisting of three different assets using the function `frontcon`. To visualize the efficient frontier curve clearly, consider 10 different evenly spaced portfolios.

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

```
ExpReturn = [0.1 0.2 0.15];

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

NumPorts = 10;
```

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

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



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

```
[PortRisk, PortReturn, PortWts] = frontcon(ExpReturn,...
ExpCovariance, NumPorts)
PortRisk =
    0.0392
    0.0445
    0.0559
    0.0701
    0.0858
    0.1023
    0.1192
    0.1383
    0.1661
    0.2000
```

```
PortReturn =

    0.1231
    0.1316
    0.1402
    0.1487
    0.1573
    0.1658
    0.1744
    0.1829
    0.1915
    0.2000

PortWts =

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

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

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

Portfolio Selection and Risk Aversion

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

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

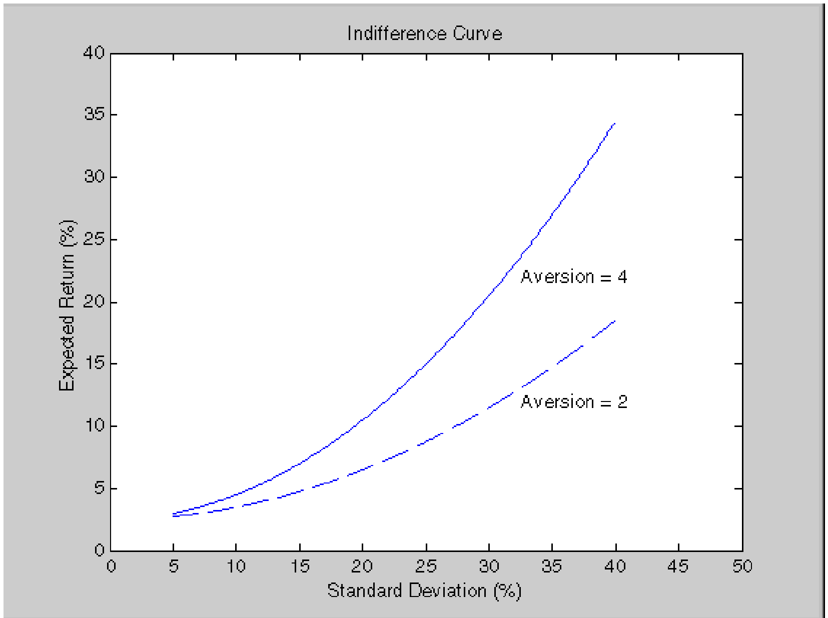
where:

`U` is the utility value.

`E(r)` is the expected return.

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

`sig` is the standard deviation.



Optimal Risky Portfolio Example

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

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

```
ExpReturn = [0.1 0.2 0.15];

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

This time consider 20 different points along the efficient frontier.

```
NumPorts = 20;
[PortRisk, PortReturn, PortWts] = portopt(ExpReturn,...
ExpCovariance, NumPorts);
```

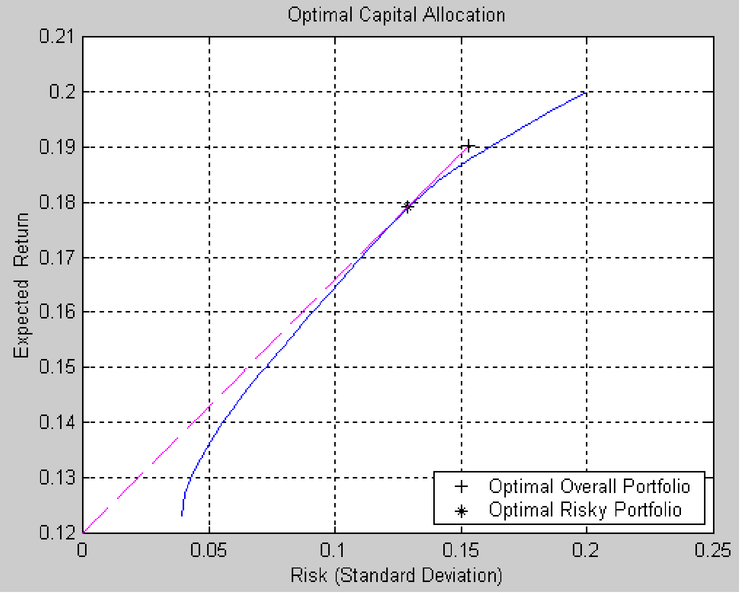
As with `frontcon`, calling `portopt` while specifying output arguments returns the corresponding vectors and arrays representing the risk, return, and weights for each of the portfolios along the efficient frontier. Use them as the first three input arguments to the function `portalloc`.

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

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

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

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



Calling `portalloc` while specifying the output arguments returns the variance (`RiskyRisk`), the expected return (`RiskyReturn`), and the weights (`RiskyWts`) allocated to the optimal risky portfolio. It also returns the fraction

(RiskyFraction) of the complete portfolio allocated to the risky portfolio, and the variance (OverallRisk) and expected return (OverallReturn) of the optimal overall portfolio. The overall portfolio combines investments in the risk-free asset and in the risky portfolio. The actual proportion assigned to each of these two investments is determined by the degree of risk aversion characterizing the investor.

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

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

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

Constraint Specification

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

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

The covariance matrix is

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

Constraint 1. Allow short selling up to 10% of the portfolio value in any asset but limit the investment in any one asset to 110% of the portfolio value.

Constraint 2. Consider two different sectors, technology and energy, with the table below indicating the sector each asset belongs to.

Asset	INTC	XON	RD
Sector	Technology	Energy	Energy

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

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

```
NumPorts = 8;
```

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

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

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

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

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

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

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

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

PortRisk =

    0.0416
    0.0499
    0.0624
    0.0767
    0.0920
    0.1100
    0.1378
    0.1716

PortReturn  =

   0.1279
   0.1361
   0.1442
   0.1524
   0.1605
   0.1687
```

```
    0.1768
    0.1850

PortWts  =

    0.7000    0.2582    0.0418
    0.6031    0.3244    0.0725
    0.4864    0.3708    0.1428
    0.3696    0.4172    0.2132
    0.2529    0.4636    0.2835
    0.2000    0.5738    0.2262
    0.2000    0.7369    0.0631
    0.2000    0.9000   -0.1000
```

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

Linear Constraint Equations

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

The auxiliary function `portcons` can be used to create the matrix of constraints, with each row representing an inequality. These inequalities are of the type `A*Wts' <= b`, where `A` is a matrix, `b` is a vector, and `Wts` is a row vector of asset allocations. The number of columns of the matrix `A`, and the length of the vector `Wts` correspond to the number of assets. The number of rows of the matrix `A`, and the length of vector `b` correspond to the number of constraints. This method allows you to specify any number of linear inequalities to the function `portopt`.

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

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

Table 3-1: Maximum and Minimum Group Exposure

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

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

Also assume that the portfolio consists of three different funds. The correspondence between funds and groups is shown in Table 3-2.

Table 3-2: Group Membership

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

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

Specifically

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

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

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

Bringing these equations into matrix notation gives

```
A = [-1    -1     0;
      1     1     0;
      0     0    -1;
      0     0     1;
     -1     0     0;
      1     0     0;
      0    -1    -1;
      0     1     1]

b = [-0.30;
      0.75;
     -0.10;
      0.55;
     -0.20;
      0.50;
     -0.50;
      0.50]
```

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

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

Specifying Additional Constraints

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

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

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

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

The only data requirement for the constraint-specifier string 'Default' is NumAssets (the total number of assets). The constraint-specifier string 'GroupLims' requires three different arguments: a Groups matrix indicating the assets that belong to each group, the GroupMin vector indicating the minimum bounds for each group, and the GroupMax vector indicating the maximum bounds for each group. Based on Table 3-2, Group Membership, build the Group matrix, with each row representing a group, and each column representing an asset.

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

Table 3-1, Maximum and Minimum Group Exposure, has the information to build GroupMin and GroupMax.

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

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

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

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

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

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

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

In this case the constraints allow only one optimum portfolio.

Active Returns and Tracking Error Efficient Frontier

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

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

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

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

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

```
NumAssets    =  5;

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

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

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

Convert the correlations and standard deviations to a covariance matrix.

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

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

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

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

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

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

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

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

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

ans =

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

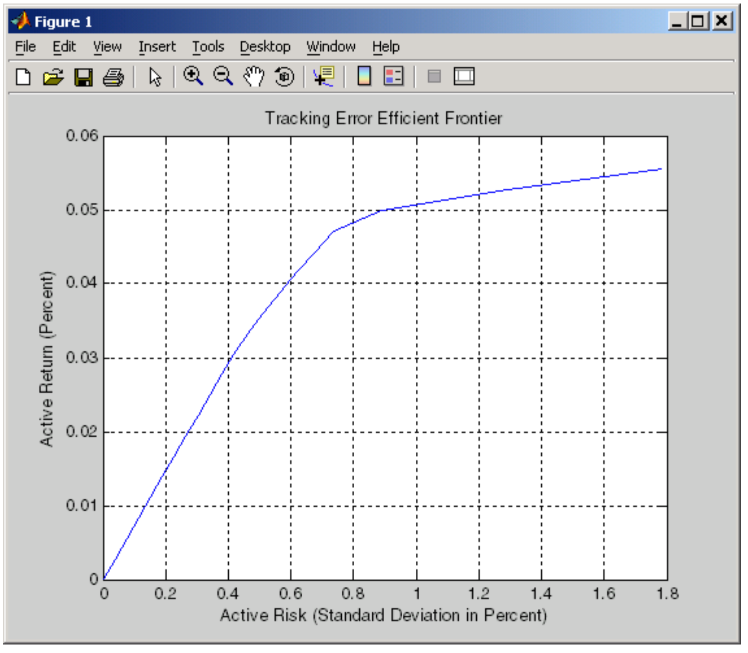
```
0      0.2000
0      0.2000
0      0.2000
```

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

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

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

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



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

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

Portfolios with Missing Data

There are times when you need to compute statistical values for portfolios, but some of the data is unavailable. The Financial Toolbox provides the function `ecmnmle`, which computes the mean and covariance of data with missing elements.

The algorithm assumes that missing values are *missing at random* and *non-ignorable*. (See Little and Rubin [1] for precise definitions of these terms.) Asset data that does not exist prior to a certain date, e.g., stock price data prior to an IPO, is an example where `ecmnmle` is appropriate. MATLAB represents these unavailable values as `NaN`. For a counterexample, consider censored data, in which all values greater than some cutoff are replaced with `NaN`s. This type of data does not satisfy the conditions under which you can use `ecmnmle`.

The general model that `ecmnmle` solves estimates the mean m and covariance C of a collection of independent identically-distributed observations of an n-dimensional multivariate normal random variable

$$Z \sim N(m, C)$$

with m observations $z(1), \ldots, z(m)$ of the random variable Z.

The collection of observations (or samples) is stored in a MATLAB matrix `Data` such that

`Data(i, :)` = $z(i)^T$

for $i = 1, \ldots, m$, where `Data` is an `m`-by-`n` matrix.

Implementation of ecmnmle

The function `ecmnmle` obtains estimates for the mean (m) and the covariance (C) of `Data` with `NumSamples = m` samples and `NumSeries = n` random variables. If data is missing, this routine implements the ECM algorithm of Meng and Rubin [2] with enhancements by Sexton and Swensen [3]. ECM stands for *expectation conditional maximization*, a conditional maximization form of the EM algorithm of Dempster, Laird, and Rubin [4].

If a record is empty, i.e., every value in a sample is `NaN`, this routine ignores the record since the record contributes no information. If such records exist in the data, the number of nonempty samples used in the estimation (*Count*) is *Count* ≤ `NumSamples`.

The estimate for the covariance is a biased maximum likelihood estimate (MLE). To formally evaluate standard errors, it is important to construct unbiased estimates. To convert to an unbiased estimate, multiply the covariance by $Count/(Count - 1)$.

Requirements

This routine has several requirements:

- Consistent values for `NumSamples` and `NumSeries` with `NumSamples > NumSeries`
- Enough nonmissing values to converge
- Positive definite covariance matrix

Although you can find some necessary and sufficient conditions in the references, general conditions for existence and uniqueness of solutions in the missing-data case do not exist. The main failure mode is an ill-conditioned covariance matrix estimate, which is discussed below in greater detail. Nonetheless, this routine works for most cases that have no more than 15% of total data with missing values (typical for most financial applications).

Technology Stock Example

This example illustrates the use of the missing data algorithm. It loads in five years of daily total return data for 12 computer technology stocks with 6 hardware and 6 software companies. The example estimates the mean and covariance matrix for these stocks, forms efficient frontiers with both a naïve approach and the ECM approach, and compares results.

You can run the example directly with `ecmtechdemo`. The steps presented here illustrate the process.

To begin the example, load in the data file.

```
load ecmtechdemo
```

This file contains three quantities:

- `Assets`: a cell array of the tickers for the 12 stocks in the example
- `Data`: a 1254-by-12 matrix of 1254 daily total returns for each of the 12 stocks
- `Dates`: a 1254-by-1 column vector of the dates associated with the data. The time period extends from April 19, 2000 to April 18, 2005.

The sixth stock in `Assets` is Google (GOOG), which started trading on August 19, 2004. Consequently, all returns prior to August 20, 2004 are missing and represented as `NaN`s. Also, Amazon (AMZN) had a few days with missing values scattered throughout the past five years.

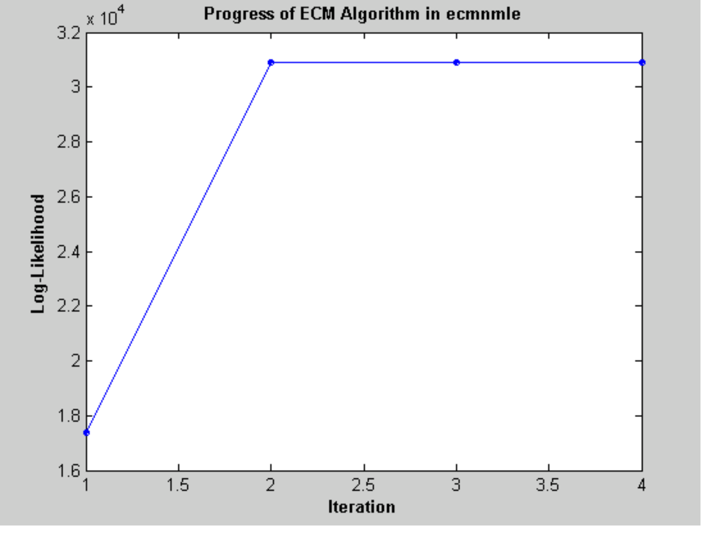
A naïve approach to the estimation of the mean and covariance for these 12 assets is to eliminate all days that have missing values for any of the 12 assets. Use the function `ecmninit` with the `nanskip` option to accomplish this.

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

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

```
ecmnmle(Data);
```

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



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

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

ECMMean =

    0.0008
    0.0008
   -0.0005
    0.0002
    0.0011
    0.0038
   -0.0003
   -0.0000
   -0.0003
   -0.0000
   -0.0003
    0.0004

ECMCovar =

    0.0012    0.0005    0.0006    0.0005    0.0005    0.0003
    0.0005    0.0024    0.0007    0.0006    0.0010    0.0004
    0.0006    0.0007    0.0013    0.0007    0.0007    0.0003
    0.0005    0.0006    0.0007    0.0009    0.0006    0.0002
    0.0005    0.0010    0.0007    0.0006    0.0016    0.0006
    0.0003    0.0004    0.0003    0.0002    0.0006    0.0022
    0.0005    0.0005    0.0006    0.0005    0.0005    0.0001
    0.0003    0.0003    0.0004    0.0003    0.0003    0.0002
    0.0006    0.0006    0.0008    0.0007    0.0006    0.0002
    0.0003    0.0004    0.0005    0.0004    0.0004    0.0001
    0.0005    0.0006    0.0008    0.0005    0.0007    0.0003
    0.0006    0.0012    0.0008    0.0007    0.0011    0.0016
    0.0005    0.0003    0.0006    0.0003    0.0005    0.0006
    0.0005    0.0003    0.0006    0.0004    0.0006    0.0012
    0.0006    0.0004    0.0008    0.0005    0.0008    0.0008
    0.0005    0.0003    0.0007    0.0004    0.0005    0.0007
    0.0005    0.0003    0.0006    0.0004    0.0007    0.0011
    0.0001    0.0002    0.0002    0.0001    0.0003    0.0016
    0.0009    0.0003    0.0005    0.0004    0.0005    0.0006
    0.0003    0.0005    0.0004    0.0003    0.0004    0.0004
    0.0005    0.0004    0.0011    0.0005    0.0007    0.0007
```

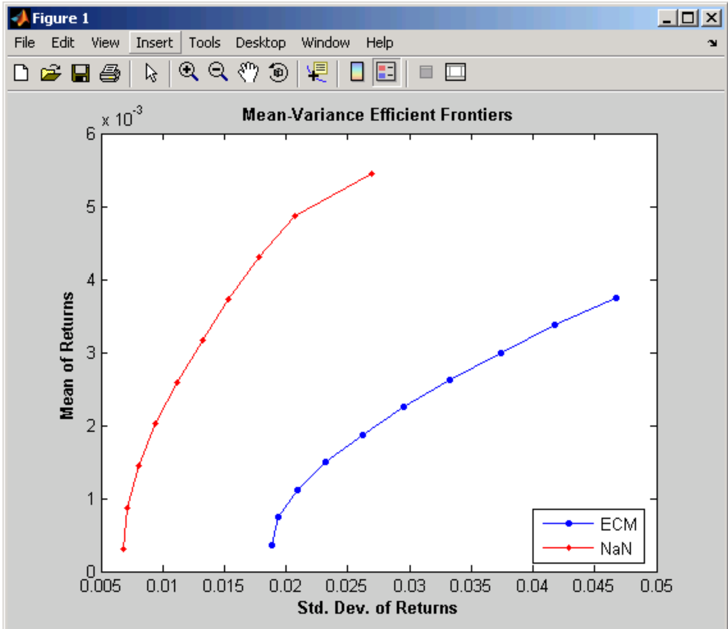
```
0.0004    0.0003    0.0005    0.0006    0.0004    0.0005
0.0005    0.0004    0.0007    0.0004    0.0013    0.0007
0.0006    0.0004    0.0007    0.0005    0.0007    0.0020
```

Given these estimates for the mean and covariance of asset returns derived from both the naïve and the ECM approaches, estimate portfolios and associated expected returns and risks on the efficient frontier for both approaches.

```
[ECMRisk, ECMReturn, ECMWts] = portopt(ECMMean',ECMCovar,10);
[NaNRisk, NaNReturn, NaNWts] = portopt(NaNMean',NaNCovar,10);
```

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

```
figure(gcf)
plot(ECMRisk, ECMReturn,'-bo','MarkerFaceColor', 'b',...
'MarkerSize', 3);
hold all
plot(NaNRisk, NaNReturn, '-r*', 'MarkerFaceColor',' r',...
'MarkerSize', 3);
title('\bfMean-Variance Efficient Frontiers');
legend('ECM','NaN','Location','SouthEast');
xlabel('\bfStd. Dev. of Returns');
ylabel('\bfMean of Returns');
hold off
```



Clearly, the naïve approach, displayed by the leftmost plot, is extremely optimistic about the risk-return tradeoffs for this universe of 12 technology stocks. The proof, however, lies in the portfolio weights. To view the weights, enter

```
Assets
ECMWts
NaNWts
```

which generates

```
Assets =

    Columns 1 through 8
      'AAPL'  'AMZN'  'CSCO'  'DELL'  'EBAY'  'GOOG'  'HPQ'  'IBM'

Columns 9 through 12
      'INTC'  'MSFT'  'ORCL'  'YHOO'
```

```
ECMWts =

Columns 1 through 8

0.0358  0.0011  -0.0000   0.0000   0.0000  0.0989  0.0535  0.4676
0.0654  0.0110   0.0000  -0.0000  -0.0000  0.1877  0.0179  0.3899
0.0923  0.0194   0.0000   0.0000  -0.0000  0.2784  0.0000  0.3025
0.1165  0.0264  -0.0000        0  -0.0000  0.3712  0.0000  0.2054
0.1407  0.0334   0.0000        0   0.0000  0.4639  0.0000  0.1083
0.1648  0.0403   0.0000        0  -0.0000  0.5566  0.0000  0.0111
0.1755  0.0457   0.0000   0.0000   0.0000  0.6532  0.0000  0.0000
0.1845  0.0509  -0.0000   0.0000        0  0.7502  0.0000 -0.0000
0.1093  0.0174  -0.0000        0        0  0.8733  0.0000 -0.0000
     0       0  -0.0000   0.0000  -0.0000  1.0000  0.0000 -0.0000

Columns 9 through 12

 0.0000    0.3431   -0.0000    0.0000
-0.0000    0.3282    0.0000   -0.0000
-0.0000    0.3074    0.0000   -0.0000
-0.0000    0.2806    0.0000   -0.0000
-0.0000    0.2538   -0.0000    0.0000
-0.0000    0.2271   -0.0000    0.0000
-0.0000    0.1255   -0.0000    0.0000
-0.0000    0.0143   -0.0000   -0.0000
-0.0000         0   -0.0000    0.0000
-0.0000   -0.0000   -0.0000    0.0000

NaNWts =

Columns 1 through 8

-0.0000  0.0000  -0.0000  0.1185  0.0000  0.0522  0.0824  0.1779
 0.0576 -0.0000  -0.0000  0.1219  0.0000  0.0854  0.1274  0.0460
 0.1248 -0.0000  -0.0000  0.0952  0.0000  0.1195  0.1674 -0.0000
 0.1969 -0.0000  -0.0000  0.0529  0.0000  0.1551  0.2056 -0.0000
 0.2690 -0.0000  -0.0000  0.0105  0.0000  0.1906  0.2438 -0.0000
 0.3414 -0.0000  -0.0000 -0.0000  0.0000  0.2265  0.2782 -0.0000
 0.4235  0.0000  -0.0000 -0.0000  0.0000  0.2639  0.2788 -0.0000
 0.5245  0.0000  -0.0000 -0.0000  0.0000  0.3034  0.1721 -0.0000
```

```
 0.6269 -0.0000  -0.0000  0.0000 -0.0000  0.3425  0.0306  0.0000
 1.0000 -0.0000  -0.0000  0.0000 -0.0000       0       0  0.0000

Columns 9 through 12

0.0000    0.5691   -0.0000    0.0000
0.0000    0.5617   -0.0000    0.0000
0.0000    0.4802    0.0129    0.0000
0.0000    0.3621    0.0274    0.0000
0.0000    0.2441    0.0419   -0.0000
0.0000    0.0988    0.0551   -0.0000
0.0000    0.0000    0.0337   -0.0000
0.0000    0.0000    0.0000   -0.0000
0.0000   -0.0000   -0.0000   -0.0000
0.0000   -0.0000   -0.0000   -0.0000
```

The naïve portfolios in `NaNWts` tend to favor Apple Computer (AAPL), which happened to do well over the period from the Google IPO to the end of the estimation period, while the ECM portfolios in `ECMWts` tend to underweight Apple and to recommend increased weights in Google relative to the naïve weights.

To evaluate the impact of estimation error and, in particular, the effect of missing data, use `ecmnstd` to calculate standard errors. Although it is possible to estimate the standard errors for both the mean and covariance, the standard errors for the mean estimates alone are usually the main quantities of interest.

The theoretical lower-bound estimate for standard errors is derived from the Fisher information matrix, computed by `ecmnstd` with the `fisher` option.

```
StdMeanF = ecmnstd(Data, ECMMean, ECMCovar, 'fisher');
```

Now, calculate standard errors that use the data-generated Hessian matrix (which accounts for the possible loss of information due to missing data). Compute this standard error by `ecmnstd` with the `hessian` option.

```
StdMeanH = ecmnstd(Data, ECMMean, ECMCovar, 'hessian');
```

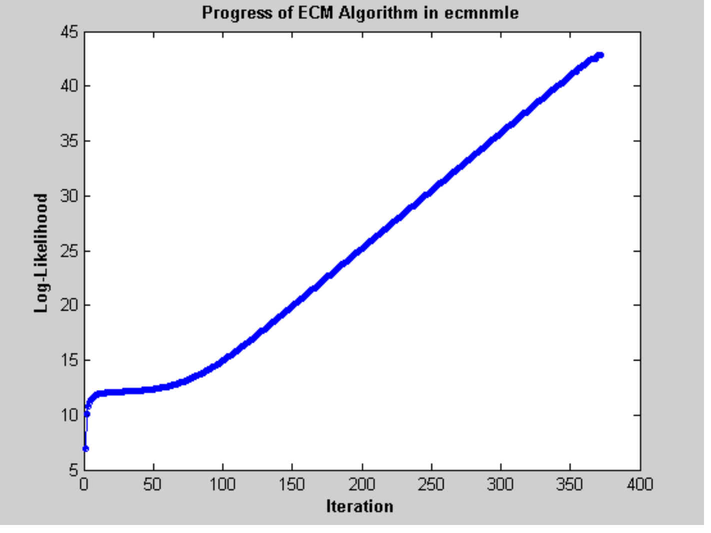
The difference in the standard errors shows the increase in uncertainty of estimation of expected returns resulting from missing data. You can view this difference by entering

```
Assets
StdMeanH'
StdMeanF'
StdMeanH' - StdMeanF'
```

The two assets with NaNs, AMZN and GOOG, are the only assets to have differences caused by the missing information.

Failure of ecmnmle

Although `ecmnmle` works for most "typical" cases, it can fail. Failures frequently derive from an ill-conditioned covariance matrix. Failures may be soft or hard. A *soft failure* randomly moves toward a very nearly singular covariance matrix. You can spot a soft failure if the algorithm fails to converge after about 100 iterations. If you increase `MaxIterations` to, say, 500 and initiate display mode (no outputs from `ecmnmle`), a typical soft failure looks like this.



This case, which is based on 20 observations of 5 assets with 30% of data missing, shows that the log-likelihood goes somewhat linearly to infinity as the likelihood function goes to zero. In this case, `ecmnmle` converges, but the covariance matrix is effectively singular with a smallest eigenvalue on the order of machine precision (`eps`).

A *hard failure* looks like this.

```
   In ecmninit at 60
   In ecmnmle at 140
??? Error using ==> ecmnmle
Full covariance not positive-definite in iteration 218.
```

From a practical standpoint, if in doubt, test the covariance matrix from `ecmnmle` to ensure that it is positive-definite, especially since a soft error has a matrix that appears to be positive-definite but actually has a near-zero-valued eigenvalue to within machine precision. To do this with a covariance estimate `Covar`, use `cond(Covar)`, where any value greater than `1/eps` is suspect.

If either type of failure occurs, however, `ecmnmle` is indicating that something is probably wrong with the data. For example, even with no missing data, two time series that are proportional have a nonpositive-definite covariance matrix.

References

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

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

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

[4] Dempster, A. P., N. M. Laird, and Donald B. Rubin, "Maximum Likelihood from Incomplete Data via the EM Algorithm," *Journal of the Royal Statistical Society*, Series B, Vol. 39, No. 1, 1977, pp. 1-37.

4

Solving Sample Problems

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

This chapter contains two major topics:

Common Problems in Finance

Sensitivity of Bond Prices to Changes in Interest Rates

Macaulay and *modified duration* measure the sensitivity of a bond's price to changes in the level of interest rates. *Convexity* measures the change in duration for small shifts in the yield curve, and thus measures the second-order price sensitivity of a bond. Both measures can gauge the vulnerability of a bond portfolio's value to changes in the level of interest rates.

Alternatively, analysts can use duration and convexity to construct a bond portfolio that is partly hedged against small shifts in the term structure. If you combine bonds in a portfolio whose duration is zero, the portfolio is insulated, to some extent, against interest rate changes. If the portfolio convexity is also zero, this insulation is even better. However, since hedging costs money or reduces expected return, you need to know how much protection results from hedging duration alone compared with hedging both duration and convexity.

This example demonstrates a way to analyze the relative importance of duration and convexity for a bond portfolio using some of the SIA-compliant bond functions in the Financial Toolbox. Using duration, it constructs a first-order approximation of the change in portfolio price to a level shift in interest rates. Then, using convexity, it calculates a second-order approximation. Finally it compares the two approximations with the true price change resulting from a change in the yield curve. The example M-file is `ftspex1.m`.

Step 1. Define three bonds using values for the settlement date, maturity date, face value, and coupon rate. For simplicity, accept default values for the coupon payment periodicity (semiannual), end-of-month payment rule (rule in effect), and day-count basis (actual/actual). Also, synchronize the coupon payment structure to the maturity date (no odd first or last coupon dates). Any inputs for which defaults are accepted are set to empty matrices (`[]`) as placeholders where appropriate.

```
Settle     = '19-Aug-1999';
Maturity   = ['17-Jun-2010'; '09-Jun-2015'; '14-May-2025'];
Face       = [100; 100; 1000];
CouponRate = [0.07; 0.06; 0.045];
```

Also, specify the yield curve information.

```
Yields = [0.05; 0.06; 0.065];
```

Step 2. Use Financial Toolbox functions to calculate the price, modified duration in years, and convexity in years of each bond.

The true price is quoted (clean) price plus accrued interest.

```
[CleanPrice, AccruedInterest] = bndprice(Yields, CouponRate,...
Settle, Maturity, 2, 0, [], [], [], [], [], Face);

Durations = bnddury(Yields, CouponRate, Settle, Maturity, 2,
0,... [], [], [], [], [], Face);

Convexities = bndconvy(Yields, CouponRate, Settle, Maturity,2,
0,... [], [], [], [], [], Face);

Prices  =  CleanPrice + AccruedInterest;
```

Step 3. Choose a hypothetical amount by which to shift the yield curve (here, 0.2 percentage point or 20 basis points).

```
dY = 0.002;
```

Weight the three bonds equally, and calculate the actual quantity of each bond in the portfolio, which has a total value of $100,000.

```
PortfolioPrice   = 100000;
PortfolioWeights = ones(3,1)/3;
PortfolioAmounts = PortfolioPrice * PortfolioWeights ./ Prices;
```

Step 4. Calculate the modified duration and convexity of the portfolio. Note that the portfolio duration or convextity is a weighted average of the durations or convexities of the individual bonds. Calculate the first- and second-order approximations of the percent price change as a function of the change in the level of interest rates.

```
PortfolioDuration  = PortfolioWeights' * Durations;
PortfolioConvexity = PortfolioWeights' * Convexities;
PercentApprox1 = -PortfolioDuration * dY * 100;
PercentApprox2 =  PercentApprox1 + ...
PortfolioConvexity*dY^2*100/2.0;
```

Step 5. Estimate the new portfolio price using the two estimates for the percent price change.

```
PriceApprox1  =  PortfolioPrice + ...
PercentApprox1 * PortfolioPrice/100;

PriceApprox2  =  PortfolioPrice + ...
PercentApprox2 * PortfolioPrice/100;
```

Step 6. Calculate the true new portfolio price by shifting the yield curve.

```
[CleanPrice, AccruedInterest] = bndprice(Yields + dY,...
CouponRate, Settle, Maturity, 2, 0, [], [], [], [], [],...
Face);

NewPrice = PortfolioAmounts' * (CleanPrice + AccruedInterest);
```

Step 7. Compare the results. The analysis results are as follows (verify these results by running the example M-file `ftspex1.m`):

- The original portfolio price was $100,000.
- The yield curve shifted up by 0.2 percentage point or 20 basis points.
- The portfolio duration and convexity are 10.3181 and 157.6346, respectively. These will be needed below for "Constructing a Bond Portfolio to Hedge Against Duration and Convexity".
- The first-order approximation, based on modified duration, predicts the new portfolio price (`PriceApprox1`) will be $97,936.37.
- The second-order approximation, based on duration and convexity, predicts the new portfolio price (`PriceApprox2`) will be $97,967.90.
- The true new portfolio price (`NewPrice`) for this yield curve shift is $97,967.51.
- The estimate using duration and convexity is quite good (at least for this fairly small shift in the yield curve), but only slightly better than the estimate using duration alone. The importance of convexity increases as the magnitude of the yield curve shift increases. Try a larger shift (`dY`) to see this effect.

The approximation formulas in this example consider only parallel shifts in the term structure, because both formulas are functions of `dY`, the change in yield.

The formulas are not well-defined unless each yield changes by the same amount. In actual financial markets, changes in yield curve level typically explain a substantial portion of bond price movements. However, other changes in the yield curve, such as slope, may also be important and are not captured here. Also, both formulas give local approximations whose accuracy deteriorates as `dY` increases in size. You can demonstrate this by running the program with larger values of `dY`.

Constructing a Bond Portfolio to Hedge Against Duration and Convexity

This example constructs a bond portfolio to hedge the portfolio of "Sensitivity of Bond Prices to Changes in Interest Rates." It assumes a long position in (holding) the portfolio, and that three other bonds are available for hedging. It chooses weights for these three other bonds in a new portfolio so that the duration and convexity of the new portfolio match those of the original portfolio. Taking a short position in the new portfolio, in an amount equal to the value of the first portfolio, partially hedges against parallel shifts in the yield curve.

Recall that portfolio duration or convexity is a weighted average of the durations or convexities of the individual bonds in a portfolio. As in the previous example, this example uses modified duration in years and convexity in years. The hedging problem therefore becomes one of solving a system of linear equations, which is very easy to do in MATLAB. The M-file for this example is `ftspex2.m`.

Step 1. Define three bonds available for hedging the original portfolio. Specify values for the settlement date, maturity date, face value, and coupon rate. For simplicity, accept default values for the coupon payment periodicity (semiannual), end-of-month payment rule (rule in effect), and day-count basis (actual/actual). Also, synchronize the coupon payment structure to the maturity date (i.e., no odd first or last coupon dates). Set any inputs for which defaults are accepted to empty matrices (`[]`) as placeholders where appropriate. The intent is to hedge against duration and convexity as well as constrain total portfolio price.

```
Settle     = '19-Aug-1999';
Maturity   = ['15-Jun-2005'; '02-Oct-2010'; '01-Mar-2025'];
Face       = [500; 1000; 250];
CouponRate = [0.07; 0.066; 0.08];
```

Also, specify the yield curve for each bond.

```
Yields = [0.06; 0.07; 0.075];
```

Step 2. Use Financial Toolbox functions to calculate the price, modified duration in years, and convexity in years of each bond.

The true price is quoted (clean price plus accrued interest.

```
[CleanPrice, AccruedInterest] = bndprice(Yields,CouponRate,...
Settle, Maturity, 2, 0, [], [], [], [], [], Face);

Durations = bnddury(Yields, CouponRate, Settle, Maturity,...
2, 0, [], [], [], [], [], Face);

Convexities = bndconvy(Yields, CouponRate, Settle,...
Maturity, 2, 0, [], [], [], [], [], Face);

Prices  =  CleanPrice + AccruedInterest;
```

Step 3. Set up and solve the system of linear equations whose solution is the weights of the new bonds in a new portfolio with the same duration and convexity as the original portfolio. In addition, scale the weights to sum to 1; that is, force them to be portfolio weights. You can then scale this unit portfolio to have the same price as the original portfolio. Recall that the original portfolio duration and convexity are 10.3181 and 157.6346, respectively. Also, note that the last row of the linear system ensures the sum of the weights is unity.

```
A = [Durations'
     Convexities'
     1 1 1];

b = [ 10.3181
     157.6346
       1];

Weights = A\b;
```

Step 4. Compute the duration and convexity of the hedge portfolio, which should now match the original portfolio.

```
PortfolioDuration  = Weights' * Durations;
PortfolioConvexity = Weights' * Convexities;
```

Step 5. Finally, scale the unit portfolio to match the value of the original portfolio and find the number of bonds required to insulate against small parallel shifts in the yield curve.

```
PortfolioValue = 100000;
HedgeAmounts   = Weights ./ Prices * PortfolioValue;
```

Step 6. Compare the results. (Verify the analysis results by running the example M-file `ftspex2.m`.)

- As required, the duration and convexity of the new portfolio are `10.3181` and `157.6346`, respectively.
- The hedge amounts for bonds 1, 2, and 3 are `-57.37`, `71.70`, and `216.27`, respectively.

Notice that the hedge matches the duration, convexity, and value ($100,000) of the original portfolio. If you are holding that first portfolio, you can hedge by taking a short position in the new portfolio.

Just as the approximations of the first example are appropriate only for small parallel shifts in the yield curve, the hedge portfolio is appropriate only for reducing the impact of small level changes in the term structure.

Sensitivity of Bond Prices to Parallel Shifts in the Yield Curve

Often bond portfolio managers want to consider more than just the sensitivity of a portfolio's price to a small shift in the yield curve, particularly if the investment horizon is long. This example shows how MATLAB can visualize the price behavior of a portfolio of bonds over a wide range of yield curve scenarios, and as time progresses toward maturity.

This example uses the Financial Toolbox bond pricing functions to evaluate the impact of time-to-maturity and yield variation on the price of a bond portfolio. It plots the portfolio value and shows the behavior of bond prices as yield and time vary. This example M-file is `ftspex3.m`.

Step 1. Specify values for the settlement date, maturity date, face value, coupon rate, and coupon payment periodicity of a four-bond portfolio. For simplicity, accept default values for the end-of-month payment rule (rule in effect) and day-count basis (actual/actual). Also, synchronize the coupon payment structure to the maturity date (no odd first or last coupon dates). Any inputs for which defaults are accepted are set to empty matrices ([]) as placeholders where appropriate.

```
Settle     = '15-Jan-1995';
Maturity   = datenum(['03-Apr-2020'; '14-May-2025'; ...
                      '09-Jun-2019'; '25-Feb-2019']);
Face       = [1000; 1000; 1000; 1000];
CouponRate = [0; 0.05; 0; 0.055];
Periods    = [0; 2; 0; 2];
```

Also, specify the points on the yield curve for each bond.

```
Yields = [0.078; 0.09; 0.075; 0.085];
```

Step 2. Use Financial Toolbox functions to calculate the true bond prices as the sum of the quoted price plus accrued interest.

```
[CleanPrice, AccruedInterest] = bndprice(Yields,...
CouponRate,Settle, Maturity, Periods,...
[], [], [], [], [], [], Face);

Prices  =  CleanPrice + AccruedInterest;
```

Step 3. Assume the value of each bond is $25,000, and determine the quantity of each bond such that the portfolio value is $100,000.

```
BondAmounts = 25000 ./ Prices;
```

Step 4. Compute the portfolio price for a rolling series of settlement dates over a range of yields. The evaluation dates occur annually on January 15, beginning on 15-Jan-1995 (settlement) and extending out to 15-Jan-2018. Thus, this step evaluates portfolio price on a grid of time of progression (dT) and interest rates (dY).

```
dy = -0.05:0.005:0.05;              % Yield changes

D  = datevec(Settle);               % Get date components
dt = datenum(D(1):2018, D(2), D(3)); % Get evaluation dates
[dT, dY]  =  meshgrid(dt, dy); % Create grid

NumTimes  =  length(dt);       % Number of time steps
NumYields =  length(dy);       % Number of yield changes
NumBonds  =  length(Maturity); % Number of bonds
```

```
% Preallocate vector
Prices = zeros(NumTimes*NumYields, NumBonds);
```

Now that the grid and price vectors have been created, compute the price of each bond in the portfolio on the grid one bond at a time.

```
for i = 1:NumBonds

   [CleanPrice, AccruedInterest] = bndprice(Yields(i)+...
   dY(:), CouponRate(i), dT(:), Maturity(i), Periods(i),...
   [], [], [], [], [], [], Face(i));

   Prices(:,i) = CleanPrice + AccruedInterest;

end
```

Scale the bond prices by the quantity of bonds.

```
Prices = Prices * BondAmounts;
```

Reshape the bond values to conform to the underlying evaluation grid.

```
Prices = reshape(Prices, NumYields, NumTimes);
```

Step 5. Plot the price of the portfolio as a function of settlement date and a range of yields, and as a function of the change in yield (`dY`). This plot illustrates the interest rate sensitivity of the portfolio as time progresses (`dT`), under a range of interest rate scenarios. With the following graphics commands, you can visualize the three-dimensional surface relative to the current portfolio value (i.e., $100,000).

```
figure                    % Open a new figure window
surf(dt, dy, Prices)      % Draw the surface
```

Add the base portfolio value to the existing surface plot.

```
hold on                   % Add the current value for reference
basemesh = mesh(dt, dy, 100000*ones(NumYields, NumTimes));
```

Make it transparent, plot it so the price surface shows through, and draw a box around the plot.

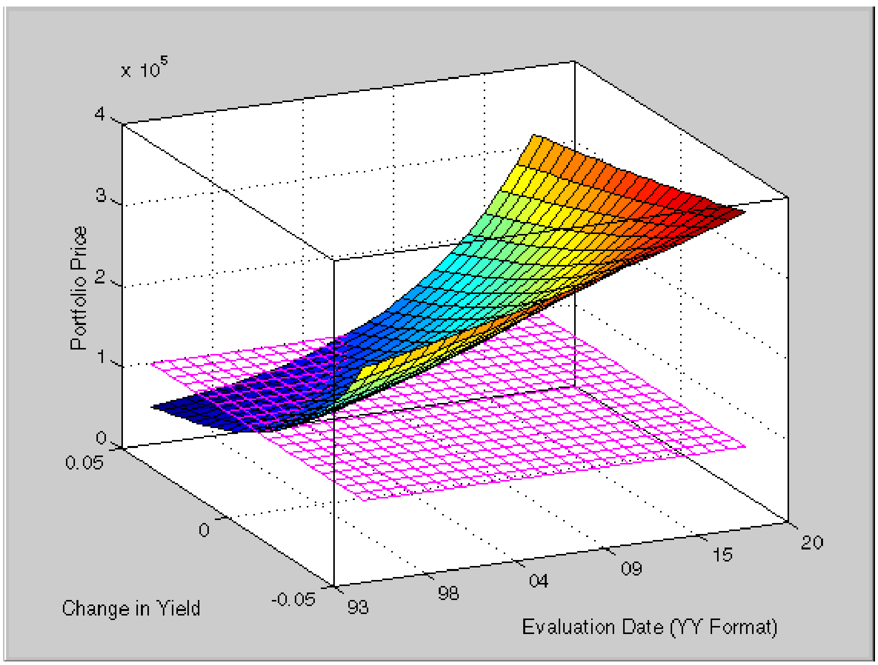
```
set(basemesh, 'facecolor', 'none');
set(basemesh, 'edgecolor', 'm');
set(gca, 'box', 'on');
```

Plot the *x*-axis using two-digit year (YY format) labels for ticks.

```
dateaxis('x', 11);
```

Add axis labels and set the three-dimensional viewpoint. MATLAB produces the figure.

```
xlabel('Evaluation Date (YY Format)');
ylabel('Change in Yield');
zlabel('Portfolio Price');
hold off
view(-25,25);
```



MATLAB three-dimensional graphics allow you to visualize the interest rate risk experienced by a bond portfolio over time. This example assumed parallel shifts in the term structure, but it might similarly have allowed other components to vary, such as the level and slope.

Constructing Greek-Neutral Portfolios of European Stock Options

The option sensitivity measures familiar to most option traders are often referred to as the *greeks*: *delta*, *gamma*, *vega*, *lambda*, *rho*, and *theta*. Delta is the price sensitivity of an option with respect to changes in the price of the underlying asset. It represents a first-order sensitivity measure analogous to duration in fixed income markets. Gamma is the sensitivity of an option's delta to changes in the price of the underlying asset, and represents a second-order price sensitivity analogous to convexity in fixed income markets. Vega is the price sensitivity of an option with respect to changes in the volatility of the underlying asset. See "Pricing and Analyzing Equity Derivatives" on page 2-33 or the "Glossary" for other definitions.

The greeks of a particular option are a function of the model used to price the option. However, given enough different options to work with, a trader can construct a portfolio with any desired values for its greeks. For example, to insulate the value of an option portfolio from small changes in the price of the underlying asset, one trader might construct an option portfolio whose delta is zero. Such a portfolio is then said to be "delta neutral." Another trader may wish to protect an option portfolio from larger changes in the price of the underlying asset, and so might construct a portfolio whose delta and gamma are both zero. Such a portfolio is both delta and gamma neutral. A third trader may wish to construct a portfolio insulated from small changes in the volatility of the underlying asset in addition to delta and gamma neutrality. Such a portfolio is then delta, gamma, and vega neutral.

Using the Black-Scholes model for European options, this example creates an equity option portfolio that is simultaneously delta, gamma, and vega neutral. The value of a particular greek of an option portfolio is a weighted average of the corresponding greek of each individual option. The weights are the quantity of each option in the portfolio. Hedging an option portfolio thus involves solving a system of linear equations, an easy process in MATLAB. This example M-file is `ftspex4.m`.

Step 1. Create an input data matrix to summarize the relevant information. Each row of the matrix contains the standard inputs to the Financial Toolbox Black-Scholes suite of functions: column 1 contains the current price of the underlying stock; column 2 the strike price of each option; column 3 the time to-expiry of each option in years; column 4 the annualized stock price volatility; and column 5 the annualized dividend rate of the underlying asset. Note that rows 1 and 3 are data related to call options, while rows 2 and 4 are data related to put options.

```
DataMatrix = [100.000  100  0.2  0.3   0        % Call
              119.100  125  0.2  0.2   0.025    % Put
               87.200   85  0.1  0.23  0        % Call
              301.125  315  0.5  0.25  0.0333]  % Put
```

Also, assume the annualized risk-free rate is 10 percent and is constant for all maturities of interest.

```
RiskFreeRate = 0.10;
```

For clarity, assign each column of `DataMatrix` to a column vector whose name reflects the type of financial data in the column.

```
StockPrice   = DataMatrix(:,1);
StrikePrice  = DataMatrix(:,2);
ExpiryTime   = DataMatrix(:,3);
Volatility   = DataMatrix(:,4);
DividendRate = DataMatrix(:,5);
```

Step 2. Based on the Black-Scholes model, compute the prices, as well as the delta, gamma, and vega sensitivity greeks of each of the four options. Note that the functions `blsprice` and `blsdelta` have two outputs, while `blsgamma` and `blsvega` have only one. The price and delta of a call option differ from the price and delta of an otherwise equivalent put option, in contrast to the gamma and vega sensitivities, which are valid for both calls and puts.

```
[CallPrices, PutPrices] = blsprice(StockPrice, StrikePrice,...
RiskFreeRate, ExpiryTime, Volatility, DividendRate);

[CallDeltas, PutDeltas] = blsdelta(StockPrice,...
StrikePrice, RiskFreeRate, ExpiryTime, Volatility,...
DividendRate);

Gammas = blsgamma(StockPrice, StrikePrice, RiskFreeRate,...
                  ExpiryTime, Volatility , DividendRate)';
```

```
Vegas  = blsvega(StockPrice, StrikePrice, RiskFreeRate,...
                 ExpiryTime, Volatility , DividendRate)';
```

Extract the prices and deltas of interest to account for the distinction between call and puts.

```
Prices = [CallPrices(1) PutPrices(2) CallPrices(3)...
PutPrices(4)];

Deltas = [CallDeltas(1) PutDeltas(2) CallDeltas(3)...
PutDeltas(4)];
```

Step 3. Now, assuming an arbitrary portfolio value of $17,000, set up and solve the linear system of equations such that the overall option portfolio is simultaneously delta, gamma, and vega-neutral. The solution computes the value of a particular greek of a portfolio of options as a weighted average of the corresponding greek of each individual option in the portfolio. The system of equations is solved using the backslash (`\`) operator discussed in "Solving Simultaneous Linear Equations" on page 1-13.

```
A = [Deltas; Gammas; Vegas; Prices];
b = [0; 0; 0; 17000];
OptionQuantities = A\b; % Quantity (number) of each option.
```

Step 4. Finally, compute the market value, delta, gamma, and vega of the overall portfolio as a weighted average of the corresponding parameters of the component options. The weighted average is computed as an inner product of two vectors.

```
PortfolioValue =  Prices * OptionQuantities;
PortfolioDelta =  Deltas * OptionQuantities;
PortfolioGamma =  Gammas * OptionQuantities;
PortfolioVega  =  Vegas  * OptionQuantities;
```

The example `ftspex4.m` performs these computations and displays the output on the screen.

```
Option  Price    Delta    Gamma    Vega     Quantity
   1   6.3441   0.5856   0.0290  17.4293   22332.6131
   2   6.6035  -0.6255   0.0353  20.0347    6864.0731
   3   4.2993   0.7003   0.0548   9.5837  -15654.8657
   4  22.7694  -0.4830   0.0074  83.5225   -4510.5153

Portfolio Value: $17000.00
Portfolio Delta:      0.00
Portfolio Gamma:     -0.00
Portfolio Vega :      0.00
```

You can verify that the portfolio value is $17,000 and that the option portfolio is indeed delta, gamma, and vega neutral, as desired. Hedges based on these measures are effective only for small changes in the underlying variables.

Term Structure Analysis and Interest Rate Swap Pricing

This example illustrates some of the term-structure analysis functions found in the Financial Toolbox. Specifically, it illustrates how to derive implied zero (*spot*) and forward curves from the observed market prices of coupon-bearing bonds. The zero and forward curves implied from the market data are then used to price an interest rate swap agreement.

In an interest rate swap, two parties agree to a periodic exchange of cash flows. One of the cash flows is based on a fixed interest rate held constant throughout the life of the swap. The other cash flow stream is tied to some variable index rate. Pricing a swap at inception amounts to finding the fixed rate of the swap agreement. This fixed rate, appropriately scaled by the notional principle of the swap agreement, determines the periodic sequence of fixed cash flows.

In general, interest rate swaps are priced from the forward curve such that the variable cash flows implied from the series of forward rates and the periodic sequence of fixed-rate cash flows have the same present value. Thus, interest rate swap pricing and term structure analysis are intimately related.

Step 1. Specify values for the settlement date, maturity dates, coupon rates, and market prices for 10 U.S. Treasury Bonds. This data allows us to price a five-year swap with net cash flow payments exchanged every six months. For simplicity, accept default values for the end-of-month payment rule (rule in effect) and day-count basis (actual/actual). To avoid issues of accrued interest, assume that all Treasury Bonds pay semiannual coupons and that settlement occurs on a coupon payment date.

```
Settle   = datenum('15-Jan-1999');

BondData = {'15-Jul-1999'  0.06000   99.93
            '15-Jan-2000'  0.06125   99.72
            '15-Jul-2000'  0.06375   99.70
            '15-Jan-2001'  0.06500   99.40
            '15-Jul-2001'  0.06875   99.73
            '15-Jan-2002'  0.07000   99.42
            '15-Jul-2002'  0.07250   99.32
            '15-Jan-2003'  0.07375   98.45
            '15-Jul-2003'  0.07500   97.71
            '15-Jan-2004'  0.08000   98.15};
```

`BondData` is an instance of a MATLAB *cell array*, indicated by the curly braces ({}).

Next assign the date stored in the cell array to `Maturity`, `CouponRate`, and `Prices` vectors for further processing.

```
Maturity   = datenum(strvcat(BondData{:,1}));
CouponRate = [BondData{:,2}]';
Prices     = [BondData{:,3}]';
Period     = 2; % semiannual coupons
```

Step 2. Now that the data has been specified, use the term structure function `zbtprice` to bootstrap the zero curve implied from the prices of the coupon-bearing bonds. This implied zero curve represents the series of zero-coupon Treasury rates consistent with the prices of the coupon-bearing bonds such that arbitrage opportunities will not exist.

```
ZeroRates = zbtprice([Maturity CouponRate], Prices, Settle);
```

The zero curve, stored in `ZeroRates`, is quoted on a semiannual bond basis (the periodic, six-month, interest rate is simply doubled to annualize). The first element of `ZeroRates` is the annualized rate over the next six months, the second element is the annualized rate over the next 12 months, and so on.

Step 3. From the implied zero curve, find the corresponding series of implied forward rates using the term structure function `zero2fwd`.

```
ForwardRates = zero2fwd(ZeroRates, Maturity, Settle);
```

The forward curve, stored in `ForwardRates`, is also quoted on a semiannual bond basis. The first element of `ForwardRates` is the annualized rate applied to the interval between settlement and six months after settlement, the second element is the annualized rate applied to the interval from six months to 12 months after settlement, and so on. This implied forward curve is also consistent with the observed market prices such that arbitrage activities will be unprofitable. Since the first forward rate is also a zero rate, the first element of `ZeroRates` and `ForwardRates` are the same.

Step 4. Now that you have derived the zero curve, convert it to a sequence of discount factors with the term structure function `zero2disc`.

```
DiscountFactors = zero2disc(ZeroRates, Maturity, Settle);
```

Step 5. From the discount factors, compute the present value of the variable cash flows derived from the implied forward rates. For plain interest rate swaps, the notional principle remains constant for each payment date and cancels out of each side of the present value equation. The next line assumes unit notional principle.

```
PresentValue = sum((ForwardRates/Period) .* DiscountFactors);
```

Step 6. Compute the swap's price (the fixed rate) by equating the present value of the fixed cash flows with the present value of the cash flows derived from the implied forward rates. Again, since the notional principle cancels out of each side of the equation, it is simply assumed to be 1.

```
SwapFixedRate = Period * PresentValue / sum(DiscountFactors);
```

The example `ftspex5.m` performs these computations and displays the output on the screen.

```
  Zero Rates  Forward Rates
    0.0614       0.0614
    0.0642       0.0670
    0.0660       0.0695
    0.0684       0.0758
    0.0702       0.0774
```

```
    0.0726        0.0846
    0.0754        0.0925
    0.0795        0.1077
    0.0827        0.1089
    0.0868        0.1239

Swap Price (Fixed Rate) = 0.0845
```

All rates are in decimal format. The swap price, 8.45%, would likely be the mid-point between a market-maker's bid/ask quotes.

Producing Graphics with the Toolbox

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

Plotting an Efficient Frontier

This example plots the efficient frontier of a hypothetical portfolio of three assets. It illustrates how to specify the expected returns, standard deviations, and correlations of a portfolio of assets, how to convert standard deviations and correlations into a covariance matrix, and how to compute and plot the efficient frontier from the returns and covariance matrix. The example also illustrates how to randomly generate a set of portfolio weights, and how to add the random portfolios to an existing plot for comparison with the efficient frontier. The example M-file is `ftgex1.m`.

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

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

Correlations = [ 1   0.8  0.4
                0.8   1   0.3
                0.4  0.3   1 ];
```

Convert the standard deviations and correlation matrix into a variance-covariance matrix with the Financial Toolbox function `corr2cov`.

```
Covariances = corr2cov(STDs, Correlations);
```

Evaluate and plot the efficient frontier at 20 points along the frontier, using the function `portopt` and the expected returns and corresponding covariance matrix. Although rather elaborate constraints can be placed on the assets in a portfolio, for simplicity accept the default constraints and scale the total value of the portfolio to 1 and constrain the weights to be positive (no short-selling).

```
portopt(Returns, Covariances, 20)
```

Now that the efficient frontier is displayed, randomly generate the asset weights for 1000 portfolios starting from the MATLAB initial state.

```
rand('state', 0)
Weights = rand(1000, 3);
```

The previous line of code generates three columns of uniformly distributed random weights, but does not guarantee they sum to 1. The following code segment normalizes the weights of each portfolio so that the total of the three weights represent a valid portfolio.

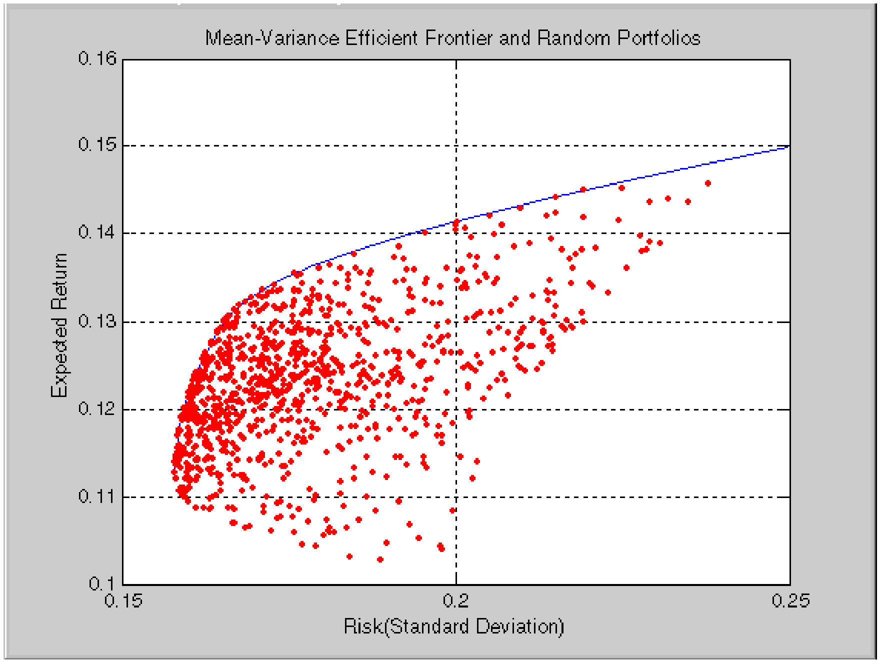
```
Total = sum(Weights, 2);     % Add the weights
Total = Total(:,ones(3,1));  % Make size-compatible matrix
Weights = Weights./Total;    % Normalize so sum = 1
```

Given the 1000 random portfolios just created, compute the expected return and risk of each portfolio associated with the weights.

```
[PortRisk, PortReturn] = portstats(Returns, Covariances, ...
                         Weights);
```

Finally, hold the current graph, and plot the returns and risks of each portfolio on top of the existing efficient frontier for comparison. After plotting, annotate the graph with a title and return the graph to default holding status (any subsequent plots will erase the existing data). The efficient frontier appears in blue, while the 1000 random portfolios appear as a set of red dots on or below the frontier.

```
hold on
plot (PortRisk, PortReturn, '.r')
title('Mean-Variance Efficient Frontier and Random Portfolios')
hold off
```



Plotting Sensitivities of an Option

This example creates a three-dimensional plot showing how gamma changes relative to price for a Black-Scholes option. Recall that gamma is the second derivative of the option price relative to the underlying security price. The plot shows a three-dimensional surface whose z-value is the gamma of an option as price (x-axis) and time (y-axis) vary. It adds yet a fourth dimension by showing option delta (the first derivative of option price to security price) as the color of the surface. This example M-file is `ftgex2.m`.

First set the price range of the options, and set the time range to one year divided into half-months and expressed as fractions of a year.

```
Range = 10:70;
Span = length(Range);
j = 1:0.5:12;
Newj = j(ones(Span,1),:)'/12;
```

For each time period create a vector of prices from 10 to 70 and create a matrix of all ones.

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

Call the toolbox gamma and delta sensitivity functions. Exercise price is $40, risk-free interest rate is 10%, and volatility is 0.35 for all prices and periods. Gamma is the *z*-axis, delta is the color.

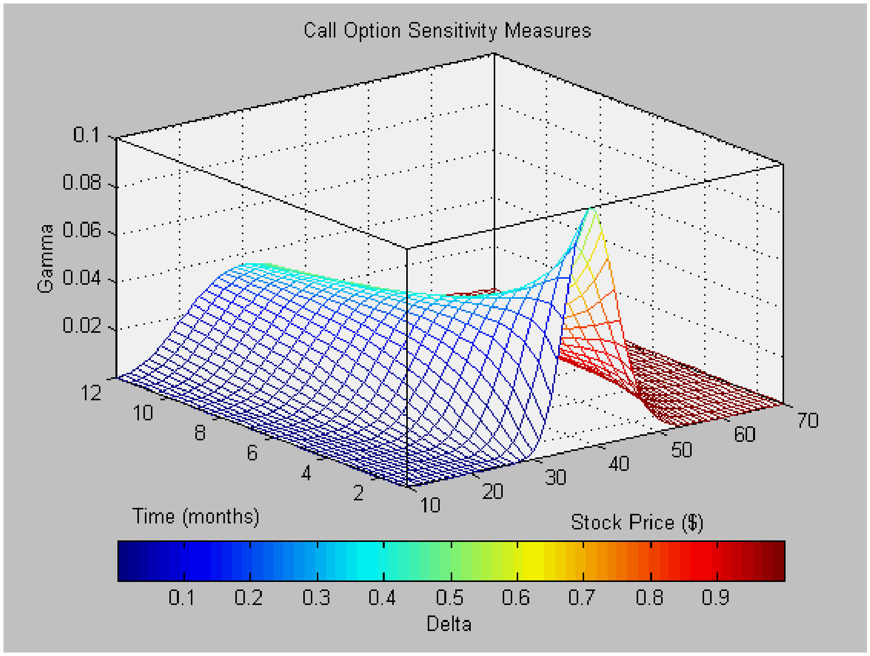
```
ZVal = blsgamma(NewRange, 40*Pad, 0.1*Pad, Newj, 0.35*Pad);
Color = blsdelta(NewRange, 40*Pad, 0.1*Pad, Newj, 0.35*Pad);
```

Draw the surface as a mesh, add axis labels and a title. The axes range from 10 to 70, 1 to 12, and -∞ to ∞.

```
mesh(Range, j, ZVal, Color);
xlabel('Stock Price ($)');
ylabel('Time (months)');
zlabel('Gamma');
title('Call Option Sensitivity Measures');
axis([10 70  1 12  -inf inf]);
```

Finally add a box around the whole plot, annotate the colors with a bar, and label the colorbar.

```
set(gca, 'box', 'on');
colorbar('horiz');
a = findobj(gcf, 'type', 'axes');
set(get(a(2), 'xlabel'), 'string', 'Delta');
```



Plotting Sensitivities of a Portfolio of Options

This example plots gamma as a function of price and time for a portfolio of 10 Black-Scholes options. The plot shows a three-dimensional surface. For each point on the surface, the height (z-value) represents the sum of the gammas for each option in the portfolio weighted by the amount of each option. The x-axis represents changing price, and the y-axis represents time. The plot adds a fourth dimension by showing delta as surface color. This example M-file is `ftgex3.m`.

First set up the portfolio with arbitrary data. Current prices range from $20 to $90 for each option. Set corresponding exercise prices for each option.

```
Range = 20:90;
PLen = length(Range);
ExPrice = [75 70 50 55 75 50 40 75 60 35];
```

Set all risk-free interest rates to 10%, and set times to maturity in days. Set all volatilities to 0.35. Set the number of options of each instrument, and allocate space for matrices.

```
Rate = 0.1*ones(10,1);
Time = [36  36  36  27  18  18  18  9  9  9];
Sigma = 0.35*ones(10,1);
NumOpt = 1000*[4  8  3  5  5.5  2  4.8  3  4.8  2.5];
ZVal = zeros(36, PLen);
Color = zeros(36, PLen);
```

For each instrument, create a matrix (of size `Time` by `PLen`) of prices for each period.

```
for i = 1:10
    Pad = ones(Time(i),PLen);
    NewR = Range(ones(Time(i),1),:);
```

Create a vector of time periods `1` to `Time`; and a matrix of times, one column for each price.

```
    T = (1:Time(i))';
    NewT = T(:,ones(PLen,1));
```

Call the toolbox gamma and delta sensitivity functions to compute gamma and delta.

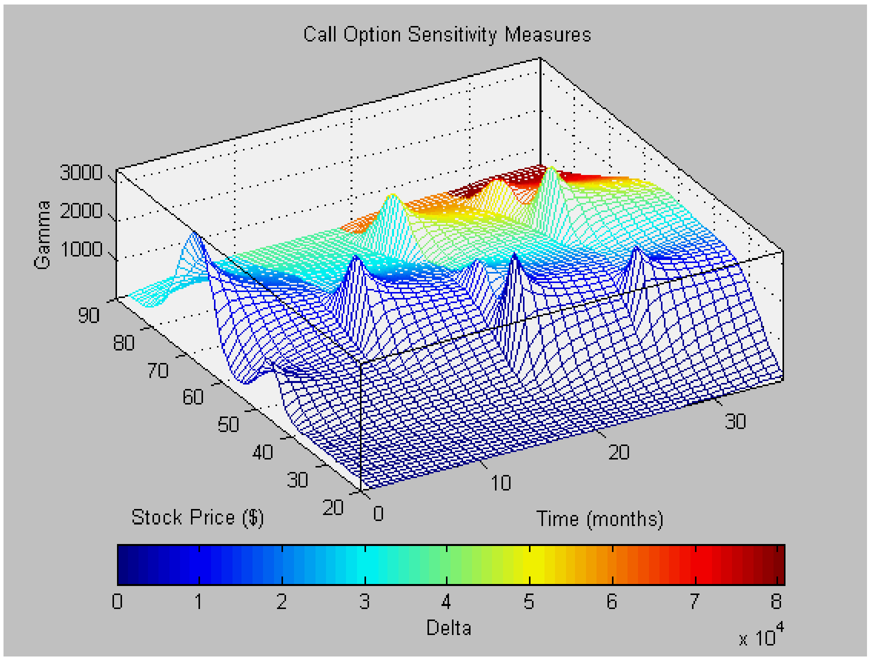
```
    ZVal(36-Time(i)+1:36,:) = ZVal(36-Time(i)+1:36,:) ...
        + NumOpt(i) * blsgamma(NewR, ExPrice(i)*Pad, ...
        Rate(i)*Pad, NewT/36, Sigma(i)*Pad);

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

Draw the surface as a mesh, set the viewpoint, and reverse the x-axis because of the viewpoint. The axes range from 20 to 90, 0 to 36, and $-\infty$ to ∞.

```
mesh(Range, 1:36, ZVal, Color);
view(60,60);
set(gca, 'xdir','reverse');
axis([20 90  0 36  -inf inf]);
```

Add a title and axis labels and draw a box around the plot. Annotate the colors with a bar and label the colorbar.

```
title('Call Option Sensitivity Measures');
xlabel('Stock Price ($)');
ylabel('Time (months)');
zlabel('Gamma');
set(gca, 'box', 'on');
colorbar('horiz');
a = findobj(gcf, 'type', 'axes');
set(get(a(2), 'xlabel'), 'string', 'Delta');
```



Function Reference

Functions - Categorical List

This chapter contains detailed descriptions of all the functions in the Financial Toolbox. The categories of functions described are:

- "Handling and Converting Dates"
- "Formatting Currency"
- "Charting Financial Data"
- "Analyzing and Computing Cash Flows"
- "Fixed-Income Securities"
- "Analyzing Portfolios"
- "Financial Statistics"
- "Pricing and Analyzing Derivatives"
- "GARCH Processes"
- "Obsolete Bond Price and Yield Functions"
- "Obsolete BDT Functions"

Handling and Converting Dates

Note The date functions `datenum`, `datestr`, `datevec`, `eomday`, `now`, and `weekday` now ship with basic MATLAB. They originally shipped only with the Financial Toolbox. Their descriptions remain in this document for your convenience.

Current Time and Date

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

Date and Time Components

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

Date Conversion

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

time2date	Dates from time and frequency
x2mdate	Excel serial date number to MATLAB serial date number.

Financial Dates

busdate		Next or previous business day.
datemnth		Date of day in future or past month.
datewrkdy		Date of future or past workday.
days360	SIA[1]	Days between dates based on 360-day year.
days360e		Days between dates based on 360-day year (European).
days360isda	ISDA[2]	Days between dates based on 360-day year.
days360psa	PSA[3]	Days between dates based on 360-day year.
days365		Days between dates based on 365-day year.
daysact		Actual number of days between dates.
daysadd		Date away from a starting date for any day-count basis
daysdif		Days between dates for any day-count basis.
fbusdate		First business date of month.
holidays		Holidays and non-trading days.
isbusday		True for dates that are business days.
lbusdate		Last business date of month.
wrkdydif		Number of working days between dates.
yearfrac		Fraction of year between dates.

[1] Securities Industry Association compliant.

[2] International Swap Dealer Association.

[3] Public Securities Association.

Coupon Bond Dates

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

Formatting Currency

`cur2frac`	Decimal currency value to fractional value.
`cur2str`	Bank formatted text.
`frac2cur`	Fractional currency value to decimal value.

Charting Financial Data

The Financial Toolbox provides a set of functions that create several of the most commonly-used types of financial charts. The Financial Time Series Toolbox provides additional charting capabilities. Using time series data as input, the Financial Time Series Toolbox can compute the value of various financial indicators and plot the results. Complete information may be found in the Financial Time Series documentation.

`bolling`	Bollinger band chart.
`candle`	Candlestick chart.
`dateaxis`	Convert serial-date axis labels to calendar-date axis labels.
`highlow`	High, low, open, close chart.
`movavg`	Leading and lagging moving averages chart.
`pointfig`	Point and figure chart.

Analyzing and Computing Cash Flows

Annuities

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

Amortization and Depreciation

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

Present Value

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

Future Value

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

Payment Calculations

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

Rates of Return

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

Cash Flow Sensitivities

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

Fixed-Income Securities

Accrued Interest

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

Prices

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

Term Structure of Interest Rates

disc2zero	Zero curve given a discount curve.
fwd2zero	Zero curve given a forward curve.
prbyzero	Price bonds in a portfolio by a set of zero curves.
pyld2zero	Zero curve given a par yield curve.
tbl2bond	Treasury bond parameters given Treasury bill parameters.
tr2bonds	Term-structure parameters given Treasury bond parameters.
zbtprice	Zero curve bootstrapping from coupon bond data given price.
zbtyield	Zero curve bootstrapping from coupon bond data given yield.
zero2disc	Discount curve given a zero curve.
zero2fwd	Forward curve given a zero curve.
zero2pyld	Par yield curve given a zero curve.

Yields

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

Spreads

`bndspread`	SIA	Static spread over spot curve

Interest Rate Sensitivities

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

Analyzing Portfolios

Portfolio Analysis

`abs2active`	Convert constraints from absolute format to active format
`active2abs`	Convert constraints from active format to absolute format
`corr2cov`	Convert standard deviation and correlation to covariance.
`cov2corr`	Convert covariance to standard deviation and correlation coefficient.
`ewstats`	Expected return and covariance from return time series.
`frontcon`	Mean-variance efficient frontier.
`pcalims`	Linear inequalities for individual asset allocation.

`pcgcomp`	Linear inequalities for asset group comparison constraints.
`pcglims`	Linear inequalities for asset group minimum and maximum allocation.
`pcpval`	Linear inequalities for fixing total portfolio value.
`portalloc`	Optimal capital allocation to efficient frontier portfolios.
`portcons`	Portfolio constraints.
`portopt`	Portfolios on constrained efficient frontier.
`portrand`	Randomized portfolio risks, returns, and weights.
`portstats`	Portfolio expected return and risk.
`portsim`	Monte Carlo simulation of correlated asset returns.
`portvrisk`	Portfolio value at risk
`ret2tick`	Convert a return series to a price series
`tick2ret`	Convert a price series to a return series

Financial Statistics

Expectation Conditional Maximization

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

Pricing and Analyzing Derivatives

Option Valuation and Sensitivity

`binprice`	Binomial put and call pricing.
`blkimpv`	Implied volatility for futures options from Black's model.
`blkprice`	Black's model for pricing futures options.
`blsdelta`	Black-Scholes sensitivity to underlying price change.
`blsgamma`	Black-Scholes sensitivity to underlying delta change.
`blsimpv`	Black-Scholes implied volatility.
`blslambda`	Black-Scholes elasticity.
`blsprice`	Black-Scholes put and call pricing.
`blsrho`	Black-Scholes sensitivity to interest rate change.
`blstheta`	Black-Scholes sensitivity to time-until-maturity change.
`blsvega`	Black-Scholes sensitivity to underlying price volatility.
`opprofit`	Option profit.

GARCH Processes

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

Univariate GARCH Processes

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

Obsolete Bond Price and Yield Functions

The functions listed in this table are obsolete, and their descriptions have been removed from the documentation. They have been replaced with the SIA-compliant functions `bndprice` and `bndyield`. For compatibility purposes, the obsolete functions remain in the product. Type `help function_name` at the MATLAB command line for a description.

Obsolete Functions

`prbond`	Price of security with regular periodic interest payments.
`proddf`	Price with odd first period.
`proddfl`	Price with odd first and last periods and settlement in first period.
`proddl`	Price with odd last period.
`yldbond`	Yield to maturity of bond.
`yldoddf`	Yield of security with odd first period.
`yldoddfl`	Yield of security with odd first and last periods and settlement in first period.
`yldoddl`	Yield of security with odd last period.

Obsolete BDT Functions

The functions `bdtbond` and `bdttrans` are obsolete, and their descriptions have been removed from the documentation. These functions have been replaced by BDT functions in the Financial Derivatives Toolbox. For compatibility purposes, the obsolete functions remain in the product. Type `help function_name` at the MATLAB command line for a description.

Functions — Alphabetical List

This section contains function reference pages listed alphabetically.

Purpose Convert constraints from absolute format to active format

Syntax

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

Arguments

`AbsConSet`	Portfolio linear inequality constraint matrix expressed in absolute weight format. `AbsConSet` is formatted as `[A b]` such that `A*w <= b`, where `A` is a number of constraints (`NCONSTRAINTS`) by number of assets (`NASSETS`) weight coefficient matrix, and `b` and `w` are column vectors of length `NASSETS`. The value `w` represents a vector of absolute asset weights whose elements sum to the total portfolio value. See the output `ConSet` from `portcons` for additional details about constraint matrices.
`Index`	`NASSETS`-by-1 vector of index portfolio weights. The sum of the index weights must equal the total portfolio value (e.g., a standard portfolio optimization imposes a sum-to-one budget constraint).

Description

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

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

Therefore

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

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

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

See Also

active2abs, pcalims, pcgcomp, pcglims, pcpval, portcons

Purpose Convert constraints from active format to absolute format

Syntax

```
AbsConSet = active2abs(ActiveConSet, Index)
```

Arguments

`ActiveConSet`	Portfolio linear inequality constraint matrix expressed in active weight format. `ActiveConSet` is formatted as `[A b]` such that `A*w <= b`, where `A` is a number of constraints (`NCONSTRAINTS`) by number of assets (`NASSETS`) weight coefficient matrix, and `b` and `w` are column vectors of length `NASSETS`. The value `w` represents a vector of active asset weights (relative to the index portfolio) whose elements sum to 0. See the output `ConSet` from `portcons` for additional details about constraint matrices.
`Index`	`NASSETS`-by-1 vector of index portfolio weights. The sum of the index weights must equal the total portfolio value (e.g., a standard portfolio optimization imposes a sum-to-one budget constraint).

Description

`AbsConSet = active2abs(ActiveConSet, Index)` transforms a constraint matrix to an equivalent matrix expressed in absolute weight format. The transformation equation is

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

Therefore

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

The initial constraint matrix consists of `NCONSTRAINTS` portfolio linear inequality constraints expressed in active weight format (relative to the index portfolio). The index portfolio vector contains `NASSETS` assets.

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

See Also abs2active, pcalims, pcgcomp, pcglims, pcpval, portcons

Purpose Fraction of coupon period before settlement (SIA compliant)

Syntax

```
Fraction = accrfrac(Settle, Maturity, Period, Basis, EndMonthRule,
   IssueDate, FirstCouponDate, LastCouponDate, StartDate)
```

Arguments

`Settle`	Settlement date. A vector of serial date numbers or date strings. `Settle` must be earlier than or equal to `Maturity`.
`Maturity`	Maturity date. A vector of serial date numbers or date strings.
`Period`	(Optional) Coupons per year of the bond. A vector of integers. Allowed values are 0, 1, 2 (default), 3, 4, 6, and 12.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. 0 = actual/actual (default), 1 = 30/360 (SIA), 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).
`EndMonthRule`	(Optional) End-of-month rule. A vector. This rule applies only when `Maturity` is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond's coupon payment date is always the same numerical day of the month. 1 = set rule on (default), meaning that a bond's coupon payment date is always the last actual day of the month.
`IssueDate`	(Optional) Date when a bond was issued.
`FirstCouponDate`	(Optional) Date when a bond makes its first coupon payment. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure.

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

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

Description

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

Examples

Given data for three bonds

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

Execute the function.

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

See Also

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

acrubond

Purpose

Accrued interest of security with periodic interest payments

Syntax

```
AccruInterest = acrubond(IssueDate, Settle, FirstCouponDate, Face,
   CouponRate, Period, Basis)
```

Arguments

`IssueDate`	Enter as serial date number or date string.
`Settle`	Enter as serial date number or date string.
`FirstCouponDate`	Enter as serial date number or date string.
`Face`	Redemption (par, face) value.
`CouponRate`	Enter as decimal fraction.
`Period`	(Optional) Coupons per year of the bond. A vector of integers. Allowed values are 0, 1, 2 (default), 3, 4, 6, and 12.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. 0 = actual/actual (default), 1 = 30/360 (SIA), 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).

Description

`AccruInterest = acrubond(IssueDate, Settle, FirstCouponDate, Face, CouponRate, Period, Basis)` returns the accrued interest for a security with periodic interest payments. This function computes the accrued interest for securities with standard, short, and long first coupon periods.

Note `cfamounts` or `accrfrac` is recommended when calculating accrued interest beyond the first period.

Examples

```
AccruInterest = acrubond('31-jan-1983', '1-mar-1993', ...
                '31-jul-1983', 100, 0.1, 2, 0)

AccruInterest =
                0.8011
```

See Also `accrfrac`, `acrudisc`, `bndprice`, `bndyield`, `cfamounts`, `datenum`

acrudisc

Purpose Accrued interest of discount security paying at maturity

Syntax

```
AccruInterest = acrudisc(Settle, Maturity, Face, Discount, Period,
   Basis)
```

Arguments

`Settle`	Enter as serial date number or date string. `Settle` must be earlier than or equal to `Maturity`.
`Maturity`	Enter as serial date number or date string.
`Face`	Redemption (par, face) value.
`Discount`	Discount rate of the security. Enter as decimal fraction.
`Period`	(Optional) Coupons per year of the bond. A vector of integers. Allowed values are 0, 1, 2 (default), 3, 4, 6, and 12.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. 0 = actual/actual (default), 1 = 30/360 (SIA), 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).

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

Examples

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

AccruInterest =
                2.0604  (or $2.06)
```

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

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

Purpose

Amortization schedule

Syntax

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

Arguments

`Rate`	Interest rate per period, as a decimal fraction.
`NumPeriods`	Number of payment periods.
`PresentValue`	Present value of the loan.
`FutureValue`	(Optional) Future value of the loan. Default = 0.
`Due`	(Optional) When payments are due: 0 = end of period (default), or 1 = beginning of period.

Description

`[Principal, Interest, Balance, Payment] = amortize(Rate, NumPeriods, PresentValue, FutureValue, Due)` returns the principal and interest payments of a loan, the remaining balance of the original loan amount, and the periodic payment.

`Principal`	Principal paid in each period. A 1-by-`NumPeriods` vector.
`Interest`	Interest paid in each period. A 1-by-`NumPeriods` vector.
`Balance`	Remaining balance of the loan in each payment period. A 1-by-`NumPeriods` vector.
`Payment`	Payment per period. A scalar.

Examples

Compute an amortization schedule for a conventional 30-year, fixed-rate mortgage with fixed monthly payments. Assume a fixed rate of 12% APR and an initial loan amount of $100,000.

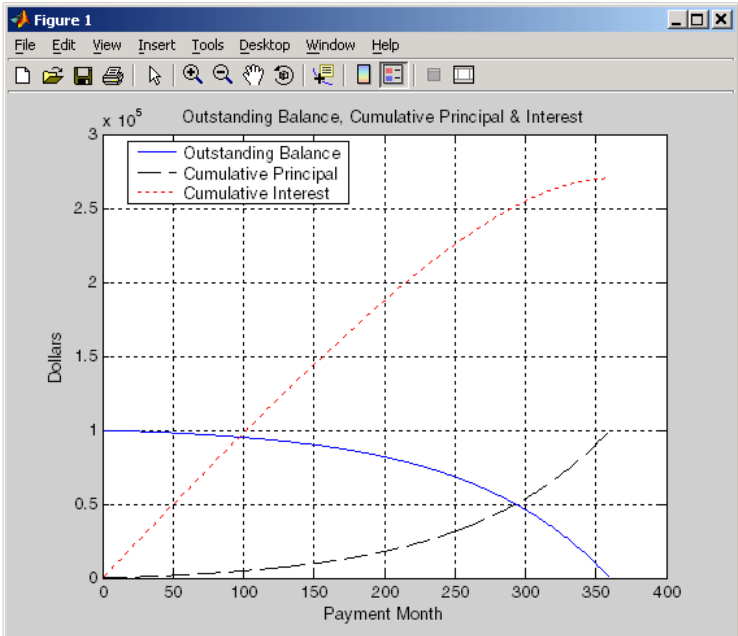
```
Rate         = 0.12/12;   % 12 percent APR = 1 percent per month
NumPeriods   = 30*12;     % 30 years = 360 months
PresentValue = 100000;

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

The output argument `Payment` contains the fixed monthly payment.

```
format bank

Payment

Payment =

       1028.61
```

Finally, summarize the amortization schedule graphically by plotting the current outstanding loan balance, the cumulative principal, and the interest payments over the life of the mortgage. In particular, note that total interest paid over the life of the mortgage exceeds $270,000, far in excess of the original loan amount!

```
plot(Balance,'b'), hold('on')
plot(cumsum(Principal),'--k')
plot(cumsum(Interest),':r')

xlabel('Payment Month')
ylabel('Dollars')
grid('on')
title('Outstanding Balance, Cumulative Principal & Interest')
legend('Outstanding Balance', 'Cumulative Principal', ...
'Cumulative Interest', 'TL')
```



The solid blue line represents the declining principal over the 30-year period. The dotted red line indicates the increasing cumulative interest payments. Finally, the dashed black line represents the cumulative principal payments, reaching $100,000 after 30 years.

See Also

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

annurate

Purpose

Periodic interest rate of annuity

Syntax

```
Rate = annurate(NumPeriods, Payment, PresentValue, FutureValue, Due)
```

Arguments

`NumPeriods`	Number of payment periods.
`Payment`	Payment per period.
`PresentValue`	Present value of the loan or annuity.
`FutureValue`	(Optional) Future value of the loan or annuity. Default = 0.
`Due`	(Optional) When payments are due: 0 = end of period (default), or 1 = beginning of period.

Description

`Rate = annurate(NumPeriods, Payment, PresentValue, FutureValue, Due)` returns the periodic interest rate paid on a loan or annuity.

Examples

Find the periodic interest rate of a four-year, $5000 loan with a $130 monthly payment made at the end of each month.

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

Rate =
      0.0094
```

(`Rate` multiplied by 12 gives an annual interest rate of 11.32% on the loan.)

See Also

`amortize`, `annuterm`, `bndyield`, `irr`

annuterm

Purpose

Number of periods to obtain value

Syntax

`NumPeriods = annuterm(Rate, Payment, PresentValue, FutureValue, Due)`

Arguments

`Rate`	Interest rate per period, as a decimal fraction.
`Payment`	Payment per period.
`PresentValue`	Present value.
`FutureValue`	(Optional) Future value. Default = 0.
`Due`	(Optional) When payments are due: 0 = end of period (default), or 1 = beginning of period.

Description

`NumPeriods = annuterm(Rate, Payment, PresentValue, FutureValue, Due)` calculates the number of periods needed to obtain a future value. To calculate the number of periods needed to pay off a loan, enter the payment or the present value as a negative value.

Examples

A savings account has a starting balance of $1500. $200 is added at the end of each month and the account pays 9% interest, compounded monthly. How many years will it take to save $5,000?

```
NumPeriods = annuterm(0.09/12, 200, 1500, 5000, 0)

NumPeriods =
            15.68  months or 1.31 years.
```

See Also

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

beytbill

Purpose Bond equivalent yield for Treasury bill

Syntax

```
Yield = beytbill(Settle, Maturity, Discount)
```

Arguments

`Settle`	Enter as serial date number or date string. `Settle` must be earlier than or equal to `Maturity`.
`Maturity`	Enter as serial date number or date string.
`Discount`	Discount rate of the Treasury bill. Enter as decimal fraction.

Description `Yield = beytbill(Settle, Maturity, Discount)` returns the bond equivalent yield for a Treasury bill.

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

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

Yield =
        0.0602
```

See Also `datenum`, `prtbill`, `yldtbill`

binprice

Purpose

Binomial put and call pricing

Syntax

```
[AssetPrice, OptionValue] = binprice(Price, Strike, Rate, Time,
   Increment, Volatility, Flag, DividendRate, Dividend, ExDiv)
```

Arguments

`Price`	Underlying asset price. A scalar.
`Strike`	Option exercise price. A scalar.
`Rate`	Risk-free interest rate. A scalar. Enter as a decimal fraction.
`Time`	Option's time until maturity in years. A scalar.
`Increment`	Time increment. A scalar. `Increment` is adjusted so that the length of each interval is consistent with the maturity time of the option. (`Increment` is adjusted so that `Time` divided by `Increment` equals an integer number of increments.)
`Volatility`	Asset's volatility. A scalar.
`Flag`	Specifies whether the option is a call (`Flag = 1`) or a put (`Flag = 0`). A scalar.
`DividendRate`	(Optional) The dividend rate, as a decimal fraction. A scalar. Default = 0. If you enter a value for `DividendRate`, set `Dividend` and `ExDiv` = `0` or do not enter them. If you enter values for `Dividend` and `ExDiv`, set `DividendRate` = `0`.
`Dividend`	(Optional) The dividend payment at an ex-dividend date, `ExDiv`. A row vector. For each dividend payment, there must be a corresponding ex-dividend date. Default = 0. If you enter values for `Dividend` and `ExDiv`, set `DividendRate` = `0`.
`ExDiv`	(Optional) Ex-dividend date, specified in number of periods. A row vector. Default = 0.

Description

`[AssetPrice, OptionValue] = binprice(Price, Strike, Rate, Time, Increment, Volatility, Flag, DividendRate, Dividend, ExDiv)` prices an option using the Cox-Ross-Rubinstein binomial pricing model.

Examples

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

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

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

```
Price =

   52.0000   58.1367   65.0226   72.7494   79.3515   89.0642
         0   46.5642   52.0336   58.1706   62.9882   70.6980
         0         0   41.7231   46.5981   49.9992   56.1192
         0         0         0   37.4120   39.6887   44.5467
         0         0         0         0   31.5044   35.3606
         0         0         0         0         0   28.0688
Option =

    4.4404    2.1627    0.6361         0         0         0
         0    6.8611    3.7715    1.3018         0         0
         0         0   10.1591    6.3785    2.6645         0
         0         0         0   14.2245   10.3113    5.4533
         0         0         0         0   18.4956   14.6394
         0         0         0         0         0   21.9312
```

See Also

`blkprice`, `blsprice`

References

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

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

Purpose

Implied volatility for futures options from Black's model

Syntax

```
Volatility = blsimpv(Price, Strike, Rate, Time, Value, Limit, ...
   Tolerance, Class)
```

Arguments

`Price`	Current price of the underlying asset (a futures contract).
`Strike`	Exercise price of the futures option.
`Rate`	Annualized, continuously compounded risk-free rate of return over the life of the option, expressed as a positive decimal number.
`Time`	Time to expiration of the option, expressed in years.
`Value`	Price of a European futures option from which the implied volatility of the underlying asset is derived.
`Limit`	(Optional) Positive scalar representing the upper bound of the implied volatility search interval. If `Limit` is empty or unspecified, the default = 10, or 1000% per annum.
`Tolerance`	(Optional) Implied volatility termination tolerance. A positive scalar. Default = 1e-6.
`Class`	(Optional) Option class (call or put) indicating the option type from which the implied volatility is derived. May be either a logical indicator or a cell array of characters. To specify call options, set `Class = true` or `Class = {'call'}`; to specify put options, set `Class = false` or `Class = {'put'}`. If `Class` is empty or unspecified, the default is a call option.

Description

`Volatility = blkimpv(Price, Strike, Rate, Time, CallPrice, MaxIterations, Tolerance)` using Black's model computes the implied volatility of a futures price from the market value of European futures options.

`Volatility` is the implied volatility of the underlying asset derived from European futures option prices, expressed as a decimal number. If no solution is found, `blkimpv` returns `NaN`.

Any input argument may be a scalar, vector, or matrix. When a value is a scalar, that value is used to compute the implied volatility of all the options. If more than one input is a vector or matrix, the dimensions of all non-scalar inputs must be identical.

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

Examples

Consider a European call futures option that expires in four months, trading at $1.1166, with an exercise price of $20. Assume that the current underlying futures price is also $20 and that the risk-free rate is 9% per annum. Furthermore, assume that you are interested in implied volatilities no greater than 0.5 (50% per annum). Under these conditions, the following commands all return an implied volatility of 0.25, or 25% per annum.

```
Volatility = blkimpv(20, 20, 0.09, 4/12, 1.1166, 0.5)
Volatility = blkimpv(20, 20, 0.09, 4/12, 1.1166,0.5,[], {'Call'})
Volatility = blkimpv(20, 20, 0.09, 4/12, 1.1166, 0.5, [], true)
```

See Also

`blkprice`, `blsimpv`, `blsprice`

References

Hull, John C., *Options, Futures, and Other Derivatives*, Prentice Hall, 5th edition, 2003, pp. 287-288.

Black, Fischer, “The Pricing of Commodity Contracts,” *Journal of Financial Economics*, March 3, 1976, pp. 167-79.

blkprice

Purpose Black's model for pricing futures options

Syntax

```
[Call, Put] = blkprice(Price, Strike, Rate, Time, Volatility)
```

Arguments

`Price`	Current price of the underlying asset (a futures contract).
`Strike`	Strike or exercise price of the futures option.
`Rate`	Annualized, continuously compounded, risk-free rate of return over the life of the option, expressed as a positive decimal number.
`Time`	Time until expiration of the option, expressed in years. Must be greater than 0.
`Volatility`	Annualized futures price volatility, expressed as a positive decimal number.

Description `[Call, Put] = blkprice(ForwardPrice, Strike, Rate, Time, Volatility)` uses Black's model to compute European put and call futures option prices.

Any input argument may be a scalar, vector, or matrix. When a value is a scalar, that value is used to compute the implied volatility from all options. If more than one input is a vector or matrix, the dimensions of all non-scalar inputs must be identical.

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

Examples Consider European futures options with exercise prices of $20 that expire in four months. Assume that the current underlying futures price is also $20 with a volatility of 25% per annum. The risk-free rate is 9% per annum. Using this data

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

returns equal call and put prices of $1.1166.

See Also `binprice`, `blsprice`

References

Hull, John C., *Options, Futures, and Other Derivatives*, Prentice Hall, 5th edition, 2003, pp. 287-288.

Black, Fischer, “The Pricing of Commodity Contracts,” *Journal of Financial Economics,* March 3, 1976, pp. 167-179.

blsdelta

Purpose

Black-Scholes sensitivity to underlying price change

Syntax

```
[CallDelta, PutDelta] = blsdelta(Price, Strike, Rate, Time,
   Volatility, Yield)
```

Arguments

`Price`	Current price of the underlying asset.
`Strike`	Exercise price of the option.
`Rate`	Annualized, continuously compounded risk-free rate of return over the life of the option, expressed as a positive decimal number.
`Time`	Time to expiration of the option, expressed in years.
`Volatility`	Annualized asset price volatility (annualized standard deviation of the continuously compounded asset return), expressed as a positive decimal number.
`Yield`	(Optional) Annualized, continuously compounded yield of the underlying asset over the life of the option, expressed as a decimal number. (Default = 0.) For example, for options written on stock indices, `Yield` could represent the dividend yield. For currency options, `Yield` could be the foreign risk-free interest rate.

Description

`[CallDelta, PutDelta] = blsdelta(Price, Strike, Rate, Time, Volatility, Yield)` returns delta, the sensitivity in option value to change in the underlying asset price. Delta is also known as the hedge ratio.

Examples

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

CallDelta =
    0.5955

PutDelta =
   -0.4045
```

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

References

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

blsgamma

Purpose

Black-Scholes sensitivity to underlying delta change

Syntax

`Gamma = blsgamma(Price, Strike, Rate, Time, Volatility, Yield)`

Arguments

`Price`	Current price of the underlying asset.
`Strike`	Exercise price of the option.
`Rate`	Annualized, continuously compounded risk-free rate of return over the life of the option, expressed as a positive decimal number.
`Time`	Time to expiration of the option, expressed in years.
`Volatility`	Annualized asset price volatility (annualized standard deviation of the continuously compounded asset return), expressed as a positive decimal number.
`Yield`	(Optional) Annualized, continuously compounded yield of the underlying asset over the life of the option, expressed as a decimal number. (Default = 0.) For example, for options written on stock indices, `Yield` could represent the dividend yield. For currency options, `Yield` could be the foreign risk-free interest rate.

Description

`Gamma = blsgamma(Price, Strike, Rate, Time, Volatility, Yield)` returns gamma, the sensitivity of delta to change in the underlying asset price.

Examples

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

Gamma =
   0.0512
```

See Also

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

References

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

blsimpv

Purpose Black-Scholes implied volatility

Syntax

```
Volatility = blsimpv(Price, Strike, Rate, Time, Value, Limit, ...
   Yield, Tolerance, Class)
```

Arguments

`Price`	Current price of the underlying asset.
`Strike`	Exercise price of the option.
`Rate`	Annualized, continuously compounded risk-free rate of return over the life of the option, expressed as a positive decimal number.
`Time`	Time to expiration of the option, expressed in years.
`Value`	Price of a European option from which the implied volatility of the underlying asset is derived.
`Limit`	(Optional) Positive scalar representing the upper bound of the implied volatility search interval. If `Limit` is empty or unspecified, the default = 10, or 1000% per annum.
`Yield`	(Optional) Annualized, continuously compounded yield of the underlying asset over the life of the option, expressed as a decimal number. (Default = 0.) For example, for options written on stock indices, `Yield` could represent the dividend yield. For currency options, `Yield` could be the foreign risk-free interest rate.
`Tolerance`	(Optional) Implied volatility termination tolerance. A positive scalar. Default = 1e-6.
`Class`	(Optional) Option class (call or put) indicating the option type from which the implied volatility is derived. May be either a logical indicator or a cell array of characters. To specify call options, set `Class = true` or `Class = {'call'}`; to specify put options, set `Class = false` or `Class = {'put'}`. If `Class` is empty or unspecified, the default is a call option.

Description

`Volatility = blsimpv(Price, Strike, Rate, Time, Value, Limit, Yield, Tolerance, Class)` using a Black-Scholes model computes the implied volatility of an underlying asset from the market value of European call and put options.

`Volatility` is the implied volatility of the underlying asset derived from European option prices, expressed as a decimal number. If no solution is found, `blsimpv` returns `NaN`.

Any input argument may be a scalar, vector, or matrix. When a value is a scalar, that value is used to price all the options. If more than one input is a vector or matrix, the dimensions of all non-scalar inputs must be identical.

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

Examples

Consider a European call option trading at $10 with an exercise price of $95 and three months until expiration. Assume that the underlying stock pays no dividend and trades at $100. The risk-free rate is 7.5% per annum. Furthermore, assume that you are interested in implied volatilities no greater than 0.5 (50% per annum).

Under these conditions, the following statements all compute an implied volatility of 0.3130, or 31.30% per annum.

```
Volatility = blsimpv(100, 95, 0.075, 0.25, 10, 0.5)
Volatility = blsimpv(100, 95, 0.075, 0.25, 10, 0.5, 0, [], {'Call'})
Volatility = blsimpv(100, 95, 0.075, 0.25, 10, 0.5, 0, [], true)
```

See Also

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

References

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

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

blslambda

Purpose Black-Scholes elasticity

Syntax

```
[CallEl, PutEl] = blslambda(Price, Strike, Rate, Time, Volatility,
   Yield)
```

Arguments

`Price`	Current price of the underlying asset.
`Strike`	Exercise price of the option.
`Rate`	Annualized, continuously compounded risk-free rate of return over the life of the option, expressed as a positive decimal number.
`Time`	Time to expiration of the option, expressed in years.
`Volatility`	Annualized asset price volatility (annualized standard deviation of the continuously compounded asset return), expressed as a positive decimal number.
`Yield`	(Optional) Annualized, continuously compounded yield of the underlying asset over the life of the option, expressed as a decimal number. (Default = 0.) For example, for options written on stock indices, `Yield` could represent the dividend yield. For currency options, `Yield` could be the foreign risk-free interest rate.

Description

`[CallEl, PutEl] = blslambda(Price, Strike, Rate, Time, Volatility, yield)` returns the elasticity of an option. `CallEl` is the call option elasticity or leverage factor, and `PutEl` is the put option elasticity or leverage factor. Elasticity (the leverage of an option position) measures the percent change in an option price per one percent change in the underlying asset price.

Examples

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

CallEl =
    8.1274

PutEl =
   -8.6466
```

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

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

blsprice

Purpose

Black-Scholes put and call option pricing

Syntax

```
[Call, Put] = blsprice(Price, Strike, Rate, Time, Volatility, Yield)
```

Arguments

`Price`	Current price of the underlying asset.
`Strike`	Exercise price of the option.
`Rate`	Annualized, continuously compounded risk-free rate of return over the life of the option, expressed as a positive decimal number.
`Time`	Time to expiration of the option, expressed in years.
`Volatility`	Annualized asset price volatility (annualized standard deviation of the continuously compounded asset return), expressed as a positive decimal number.
`Yield`	(Optional) Annualized, continuously compounded yield of the underlying asset over the life of the option, expressed as a decimal number. (Default = 0.) For example, for options written on stock indices, `Yield` could represent the dividend yield. For currency options, `Yield` could be the foreign risk-free interest rate.

Description

`[Call, Put] = blsprice(Price, Strike, Rate, Time, Volatility, Yield)` computes European put and call option prices using a Black-Scholes model.

Any input argument may be a scalar, vector, or matrix. When a value is a scalar, that value is used to price all the options. If more than one input is a vector or matrix, the dimensions of all non-scalar inputs must be identical.

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

Examples

Consider European stock options that expire in three months with an exercise price of $95. Assume that the underlying stock pays no dividend, trades at $100, and has a volatility of 50% per annum. The risk-free rate is 10% per annum. Using this data

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

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

See Also

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

References

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

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

blsrho

Purpose Black-Scholes sensitivity to interest rate change

Syntax

```
[CallRho, PutRho]= blsrho(Price, Strike, Rate, Time, Volatility,
   Yield)
```

Arguments

`Price`	Current price of the underlying asset.
`Strike`	Exercise price of the option.
`Rate`	Annualized, continuously compounded risk-free rate of return over the life of the option, expressed as a positive decimal number.
`Time`	Time to expiration of the option, expressed in years.
`Volatility`	Annualized asset price volatility (annualized standard deviation of the continuously compounded asset return), expressed as a positive decimal number.
`Yield`	(Optional) Annualized, continuously compounded yield of the underlying asset over the life of the option, expressed as a decimal number. (Default = 0.) For example, for options written on stock indices, `Yield` could represent the dividend yield. For currency options, `Yield` could be the foreign risk-free interest rate.

Description `[CallRho, PutRho]= blsrho(Price, Strike, Rate, Time, Volatility, Yield)` returns the call option rho `CallRho`, and the put option rho `PutRho`. Rho is the rate of change in value of derivative securities with respect to interest rates.

Examples

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

CallRho =
    6.6686

PutRho =
   -5.4619
```

See Also

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

References

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

blstheta

Purpose Black-Scholes sensitivity to time-until-maturity change

Syntax

```
[CallTheta, PutTheta] = blstheta(Price, Strike, Rate, Time,
   Volatility, Yield)
```

Arguments

`Price`	Current price of the underlying asset.
`Strike`	Exercise price of the option.
`Rate`	Annualized, continuously compounded risk-free rate of return over the life of the option, expressed as a positive decimal number.
`Time`	Time to expiration of the option, expressed in years.
`Volatility`	Annualized asset price volatility (annualized standard deviation of the continuously compounded asset return), expressed as a positive decimal number.
`Yield`	(Optional) Annualized, continuously compounded yield of the underlying asset over the life of the option, expressed as a decimal number. (Default = 0.) For example, for options written on stock indices, `Yield` could represent the dividend yield. For currency options, `Yield` could be the foreign risk-free interest rate.

Description `[CallTheta, PutTheta] = blstheta(Price, Strike, Rate, Time, Volatility, Yield)` returns the call option theta `CallTheta`, and the put option theta `PutTheta`. Theta is the sensitivity in option value with respect to time.

Examples

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

CallTheta =
   -8.9630

PutTheta =
   -3.1404
```

See Also

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

References

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

blsvega

Purpose

Black-Scholes sensitivity to underlying price volatility

Syntax

```
Vega = blsvega(Price, Strike, Rate, Time, Volatility, Yield)
```

Arguments

`Price`	Current price of the underlying asset.
`Strike`	Exercise price of the option.
`Rate`	Annualized, continuously compounded risk-free rate of return over the life of the option, expressed as a positive decimal number.
`Time`	Time to expiration of the option, expressed in years.
`Volatility`	Annualized asset price volatility (annualized standard deviation of the continuously compounded asset return), expressed as a positive decimal number.
`Yield`	(Optional) Annualized, continuously compounded yield of the underlying asset over the life of the option, expressed as a decimal number. (Default = 0.) For example, for options written on stock indices, `Yield` could represent the dividend yield. For currency options, `Yield` could be the foreign risk-free interest rate.

Description

`Vega = blsvega(Price, Strike, Rate, Time, Volatility, Yield)` returns vega, the rate of change of the option value with respect to the volatility of the underlying asset.

Examples

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

Vega =
    9.6035
```

See Also

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

References

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

Purpose

Bond convexity given price (SIA compliant)

Syntax

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

Arguments

`Price`	Clean price (excludes accrued interest).
`CouponRate`	Decimal number indicating the annual percentage rate used to determine the coupons payable on a bond.
`Settle`	Settlement date. A vector of serial date numbers or date strings. `Settle` must be earlier than or equal to `Maturity`.
`Maturity`	Maturity date. A vector of serial date numbers or date strings.
`Period`	(Optional) Coupons per year of the bond. A vector of integers. Allowed values are 0, 1, 2 (default), 3, 4, 6, and 12.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. 0 = actual/actual (default), 1 = 30/360 (SIA), 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).
`EndMonthRule`	(Optional) End-of-month rule. A vector. This rule applies only when `Maturity` is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond's coupon payment date is always the same numerical day of the month. 1 = set rule on (default), meaning that a bond's coupon payment date is always the last actual day of the month.
`IssueDate`	(Optional) Date when a bond was issued.
`FirstCouponDate`	(Optional) Date when a bond makes its first coupon payment. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure.

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

All specified arguments must be number of bonds (`NUMBONDS`) by 1 or 1-by-`NUMBONDS` conforming vectors or scalar arguments. Use an empty matrix (`[]`) as a placeholder for an optional argument. Fill unspecified entries in input vectors with `NaN`. Dates can be serial date numbers or date strings.

Description

`[YearConvexity, PerConvexity] = bndconvp(Price, CouponRate, Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate, StartDate, Face)` computes the convexity of `NUMBONDS` fixed income securities given a clean price for each bond. This function determines the convexity for a bond whether or not the first or last coupon periods in the coupon structure are short or long (i.e., whether or not the coupon structure is synchronized to maturity). This function also determines the convexity of a zero coupon bond.

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

Examples

Find the convexity of three bonds given their prices.

```
Price = [106; 100; 98];
CouponRate = 0.055;
Settle = '02-Aug-1999';
Maturity = '15-Jun-2004';
Period = 2;
Basis = 0;

[YearConvexity, PerConvexity] = bndconvp(Price,...
CouponRate,Settle, Maturity, Period, Basis)

YearConvexity =

   21.4447
   21.0363
   20.8951

PerConvexity =

   85.7788
   84.1454
   83.5803
```

See Also

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

bndconvy

Purpose

Bond convexity given yield (SIA compliant)

Syntax

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

Arguments

`Yield`	Yield to maturity on a semiannual basis.
`CouponRate`	Decimal number indicating the annual percentage rate used to determine the coupons payable on a bond.
`Settle`	Settlement date. A vector of serial date numbers or date strings. `Settle` must be earlier than or equal to `Maturity`.
`Maturity`	Maturity date. A vector of serial date numbers or date strings.
`Period`	(Optional) Coupons per year of the bond. A vector of integers. Allowed values are 0, 1, 2 (default), 3, 4, 6, and 12.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. 0 = actual/actual (default), 1 = 30/360 (SIA), 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).
`EndMonthRule`	(Optional) End-of-month rule. A vector. This rule applies only when `Maturity` is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond's coupon payment date is always the same numerical day of the month. 1 = set rule on (default), meaning that a bond's coupon payment date is always the last actual day of the month.
`IssueDate`	(Optional) Date when a bond was issued.
`FirstCouponDate`	(Optional) Date when a bond makes its first coupon payment. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure.

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

All specified arguments must be number of bonds (`NUMBONDS`) by 1 or 1-by-`NUMBONDS` conforming vectors or scalar arguments. Use an empty matrix (`[]`) as a placeholder for an optional argument. Fill unspecified entries in input vectors with `NaN`. Dates can be serial date numbers or date strings.

Description

`[YearConvexity, PerConvexity] = bndconvy(Yield, CouponRate, Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate, StartDate, Face)` computes the convexity of `NUMBONDS` fixed income securities given the yield to maturity for each bond. This function determines the convexity for a bond whether or not the first or last coupon periods in the coupon structure are short or long (i.e., whether or not the coupon structure is synchronized to maturity). This function also determines the convexity of a zero coupon bond.

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

Examples

Find the convexity of a bond at three different yield values.

```
Yield = [0.04; 0.055; 0.06];
CouponRate = 0.055;
Settle = '02-Aug-1999';
Maturity = '15-Jun-2004';
Period = 2;
Basis = 0;

[YearConvexity, PerConvexity]=bndconvy(Yield, CouponRate,...
Settle, Maturity, Period, Basis)

YearConvexity =

   21.4825
   21.0358
   20.8885

PerConvexity =

   85.9298
   84.1434
   83.5541
```

See Also

`bndconvp`, `bnddurp`, `bnddury`, `cfconv`, `cfdur`

Purpose

Bond duration given price (SIA compliant)

Syntax

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

Arguments

`Price`	Clean price (excludes accrued interest).
`CouponRate`	Decimal number indicating the annual percentage rate used to determine the coupons payable on a bond.
`Settle`	Settlement date. A vector of serial date numbers or date strings. `Settle` must be earlier than or equal to `Maturity`.
`Maturity`	Maturity date. A vector of serial date numbers or date strings.
`Period`	(Optional) Coupons per year of the bond. A vector of integers. Allowed values are 0, 1, 2 (default), 3, 4, 6, and 12.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. 0 = actual/actual (default), 1 = 30/360 (SIA), 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).
`EndMonthRule`	(Optional) End-of-month rule. A vector. This rule applies only when `Maturity` is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond's coupon payment date is always the same numerical day of the month. 1 = set rule on (default), meaning that a bond's coupon payment date is always the last actual day of the month.
`IssueDate`	(Optional) Date when a bond was issued.
`FirstCouponDate`	(Optional) Date when a bond makes its first coupon payment. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure.

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

All specified arguments must be number of bonds (NUMBONDS) by 1 or 1-by-NUMBONDS conforming vectors or scalar arguments. Use an empty matrix ([]) as a placeholder for an optional argument. Fill unspecified entries in input vectors with NaN. Dates can be serial date numbers or date strings.

Description

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

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

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

Examples

Find the duration of three bonds given their prices.

```
Price = [106; 100; 98];
CouponRate = 0.055;
Settle = '02-Aug-1999';
Maturity = '15-Jun-2004';
Period = 2;
Basis = 0;

[ModDuration, YearDuration, PerDuration] = bnddurp(Price,...
CouponRate, Settle, Maturity, Period, Basis)

ModDuration =

    4.2400
    4.1925
    4.1759

YearDuration =

    4.3275
    4.3077
    4.3007

PerDuration =

    8.6549
    8.6154
    8.6014
```

See Also

bndconvp, bndconvy, bnddury

bnddury

Purpose

Bond duration given yield (SIA compliant)

Syntax

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

Arguments

`Yield`	Yield to maturity on a semiannual basis.
`CouponRate`	Decimal number indicating the annual percentage rate used to determine the coupons payable on a bond.
`Settle`	Settlement date. A vector of serial date numbers or date strings. `Settle` must be earlier than or equal to `Maturity`.
`Maturity`	Maturity date. A vector of serial date numbers or date strings.
`Period`	(Optional) Coupons per year of the bond. A vector of integers. Allowed values are 0, 1, 2 (default), 3, 4, 6, and 12.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. 0 = actual/actual (default), 1 = 30/360 (SIA), 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).
`EndMonthRule`	(Optional) End-of-month rule. A vector. This rule applies only when `Maturity` is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond's coupon payment date is always the same numerical day of the month. 1 = set rule on (default), meaning that a bond's coupon payment date is always the last actual day of the month.
`IssueDate`	(Optional) Date when a bond was issued.
`FirstCouponDate`	(Optional) Date when a bond makes its first coupon payment. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure.

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

All specified arguments must be number of bonds (NUMBONDS) by 1 or 1-by-NUMBONDS conforming vectors or scalar arguments. Use an empty matrix ([]) as a placeholder for an optional argument. Fill unspecified entries in input vectors with NaN. Dates can be serial date numbers or date strings.

Description

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

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

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

Examples

Find the duration of a bond at three different yield values.

```
Yield = [0.04; 0.055; 0.06];
CouponRate = 0.055;
Settle = '02-Aug-1999';
Maturity = '15-Jun-2004';
Period = 2;
Basis = 0;

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

ModDuration =

    4.2444
    4.1924
    4.1751

YearDuration =

    4.3292
    4.3077
    4.3004

PerDuration =

    8.6585
    8.6154
    8.6007
```

See Also

bndconvp, bndconvy, bnddurp

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

Syntax

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

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

`Yield`	Bond yield to maturity on a semiannual basis.
`CouponRate`	Decimal number indicating the annual percentage rate used to determine the coupons payable on a bond.
`Settle`	Settlement date. A vector of serial date numbers or date strings. `Settle` must be earlier than or equal to `Maturity`.
`Maturity`	Maturity date. A vector of serial date numbers or date strings.
`Period`	(Optional) Coupons per year of the bond. A vector of integers. Allowed values are 0, 1, 2 (default), 3, 4, 6, and 12.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. 0 = actual/actual (default), 1 = 30/360 (SIA), 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).
`EndMonthRule`	(Optional) End-of-month rule. A vector. This rule applies only when `Maturity` is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond's coupon payment date is always the same numerical day of the month. 1 = set rule on (default), meaning that a bond's coupon payment date is always the last actual day of the month.

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

Description

`[Price, AccruedInt] = bndprice(Yield, CouponRate, Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate, StartDate, Face)` given bonds with SIA date parameters and semiannual yields to maturity, returns the clean prices and accrued interest due.

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

`AccruedInt` is the accrued interest payable at settlement.

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

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

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

Examples

Price a treasury bond at three different yield values.

```
Yield = [0.04; 0.05; 0.06];
CouponRate = 0.05;
Settle = '20-Jan-1997';
Maturity = '15-Jun-2002';
Period = 2;
Basis = 0;

[Price, AccruedInt] = bndprice(Yield, CouponRate, Settle,...
Maturity, Period, Basis)

Price =

  104.8106
   99.9951
   95.4384

AccruedInt =

  0.4945
  0.4945
  0.4945
```

See Also

`cfamounts`, `bndyield`

bndspread

Purpose Static spread over spot curve

Syntax

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

Arguments

`SpotInfo`	Two-column matrix: `[SpotDates ZeroRates]` Zero rates correspond to maturities on the spot dates, continuously compounded. You will obtain the best results if you choose evenly spaced rates close together, for example, by using the three-month deposit rates.
`Price`	Price for every $100 notional amount of bonds whose spreads are computed.
`CouponRate`	Decimal number indicating the annual percentage rate used to determine the coupons payable on a bond.
`Settle`	Settlement date. A vector of serial date numbers or date strings. `Settle` must be earlier than or equal to `Maturity`.
`Maturity`	Maturity date. A vector of serial date numbers or date strings.
`Period`	(Optional) Coupons per year of the bond. A scalar or vector of integers. Allowed values are 0, 1, 2 (default), 3, 4, 6, and 12.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. 0 = actual/actual (default), 1 = 30/360 (SIA), 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).

EndMonthRule	(Optional) End-of-month rule. A vector. This rule applies only when Maturity is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond's coupon payment date is always the same numerical day of the month. 1 = set rule on (default), meaning that a bond's coupon payment date is always the last actual day of the month.
IssueDate	(Optional) Date when a bond was issued.
FirstCouponDate	(Optional) Date when a bond makes its first coupon payment. When FirstCouponDate and LastCouponDate are both specified, FirstCouponDate takes precedence in determining the coupon payment structure.
LastCouponDate	(Optional) Last coupon date of a bond prior to the maturity date. In the absence of a specified FirstCouponDate, a specified LastCouponDate determines the coupon structure of the bond. The coupon structure of a bond is truncated at the LastCouponDate regardless of where it falls and will be followed only by the bond's maturity cash flow date.

Description

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

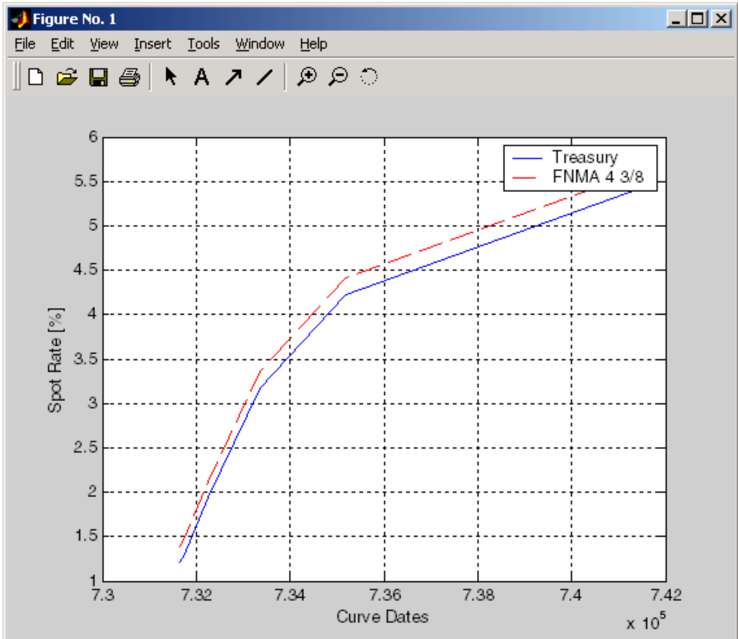
Examples

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

```
% Build spot curve.

RefMaturity = [datenum('02/27/2003');
               datenum('05/29/2003');
               datenum('10/31/2004');
               datenum('11/15/2007');
               datenum('11/15/2012');
               datenum('02/15/2031')];

RefCpn = [0;
          0;
```

```
          2.125;
          3;
          4;
          5.375] / 100;

RefPrices =  [99.6964;
               99.3572;
              100.3662;
               99.4511;
               99.4299;
              106.5756];

RefBonds = [RefPrices, RefMaturity, RefCpn];
Settle   = datenum('26-Nov-2002');
[ZeroRates, CurveDates] = zbtprice(RefBonds(:, 2:end), ...
RefPrices, Settle)

% FNMA 4 3/8 maturing 10/06 at 4.30 pm Tuesday, Nov 26, 2002
Price    = 105.484;
Coupon   = 0.04375;
Maturity = datenum('15-Oct-2006');

% All optional inputs are supposed to be accounted by default,
% except the accrued interest under 30/360 (SIA), so:
Period = 2;
Basis  = 1;
SpotInfo = [CurveDates, ZeroRates];

% Compute static spread over treasury curve, taking into account
% the shape of curve as derived by bootstrapping method embedded
% within bndspread.

SpreadInBP = bndspread(SpotInfo, Price, Coupon, Settle, ...
Maturity, Period, Basis)

plot(CurveDates, ZeroRates*100, 'b', CurveDates, ...
ZeroRates*100+SpreadInBP/100, 'r--')
legend({'Treasury'; 'FNMA 4 3/8'})
xlabel('Curve Dates')
ylabel('Spot Rate [%]')
```

```
grid;

ZeroRates =

    0.0121
    0.0127
    0.0194
    0.0317
    0.0423
    0.0550

CurveDates =

      731639
      731730
      732251
      733361
      735188
      741854

SpreadInBP =

   18.7582
```



See Also

`bndprice`, `bndyield`

Purpose

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

Syntax

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

Arguments

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

`Price`	Clean price of the bond (current price without accrued interest).
`CouponRate`	Decimal number indicating the annual percentage rate used to determine the coupons payable on a bond.
`Settle`	Settlement date. A vector of serial date numbers or date strings. `Settle` must be earlier than or equal to `Maturity`.
`Maturity`	Maturity date. A vector of serial date numbers or date strings.
`Period`	(Optional) Coupons per year of the bond. A vector of integers. Allowed values are 0, 1, 2 (default), 3, 4, 6, and 12.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. 0 = actual/actual (default), 1 = 30/360 (SIA), 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).
`EndMonthRule`	(Optional) End-of-month rule. A vector. This rule applies only when `Maturity` is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond's coupon payment date is always the same numerical day of the month. 1 = set rule on (default), meaning that a bond's coupon payment date is always the last actual day of the month.

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

Description

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

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

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

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

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

Examples

Compute the yield of a treasury bond at three different price values.

```
Price = [95; 100; 105];
CouponRate = 0.05;
Settle = '20-Jan-1997';
Maturity = '15-Jun-2002';
Period = 2;
Basis = 0;

Yield = bndyield(Price, CouponRate, Settle,...
Maturity, Period, Basis)

Yield =

    0.0610
    0.0500
    0.0396
```

See Also

`bndprice`, `cfamounts`

bolling

Purpose

Bollinger band chart

Syntax

```
bolling(Asset, Samples, Alpha)
[Movavgv, UpperBand, LowerBand] = bolling(Asset, Samples, Alpha,
   Width)
```

Arguments

`Asset`	Vector of asset data.
`Samples`	Number of samples to use in computing the moving average.
`Alpha`	(Optional) Exponent used to compute the element weights of the moving average. Default = 0 (simple moving average).
`Width`	(Optional) Number of standard deviations to include in the envelope. A multiplicative factor specifying how tight the bands should be around the simple moving average. Default = 2.

Description

`bolling(Asset, Samples, Alpha, Width)` plots Bollinger bands for given `Asset` data. This form of the function does not return any data.

`[Movavgv, UpperBand, LowerBand] = bolling(Asset, Samples, Alpha, Width)` returns `Movavgv` with the moving average of the `Asset` data, `UpperBand` with the upper band data, and `LowerBand` with the lower band data. This form of the function does not plot any data.

Note The standard deviations are normalized by N-1, where N = the sequence length.

Examples

If `Asset` is a column vector of closing stock prices

```
bolling(Asset, 20, 1)
```

plots linear 20-day moving average Bollinger bands based on the stock prices.

```
[Movavgv, UpperBand, LowerBand] = bolling(Asset, 20, 1)
```

returns `Movavgv`, `UpperBand`, and `LowerBand` as vectors containing the moving average, upper band, and lower band data, without plotting the data.

See Also `candle`, `dateaxis`, `highlow`, `movavg`, `pointfig`

busdate

Purpose

Next or previous business day

Syntax

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

Arguments

`Date`	Reference date. Enter as serial date number or date string.
`Direction`	(Optional) Direction. 1 = next (default) or -1 = previous business day.
`Holiday`	(Optional) Vector of holidays and nontrading-day dates. All dates in `Holiday` must be the same format: either serial date numbers or date strings. (Using serial date numbers improves performance.) The `holidays` function supplies the default vector.
`Weekend`	(Optional) Vector of length 7, containing 0 and 1, the value 1 indicating weekend days. The first element of this vector corresponds to Sunday. Thus, when Saturday and Sunday form the weekend (default), `Weekend = [1 0 0 0 0 0 1]`.

Description

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

Use the function `datestr` to convert serial date numbers to formatted date strings.

Examples

Example 1:

```
Busday = busdate('3-Jul-2001', 1)
Busday =

     731037

datestr(Busday)

ans =

05-Jul-2001
```

Example 2: You can indicate that Saturday is a business day by appropriately setting the `Weekend` argument.

```
Weekend = [1 0 0 0 0 0 0];
```

July 4, 2003, falls on a Friday. Use `busdate` to verify that Saturday, July 5, is actually a business day.

```
Date = datestr(busdate('3-Jul-2001', 1, , Weekend))
```

See Also

`holidays`, `isbusday`

candle

Purpose

Candlestick chart

Syntax

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

Arguments

`High`	High prices for a security. A column vector.
`Low`	Low prices for a security. An column vector.
`Close`	Closing prices for a security. A column vector.
`Open`	Opening prices for a security. A column vector.
`Color`	(Optional) Candlestick color. A string. MATLAB supplies a default color if none is specified. The default color differs depending on the background color of the figure window. See `ColorSpec` in the MATLAB documentation for color names.

Description

`candle(High, Low, Close, Open, Color)` plots a candlestick chart given column vectors with the high, low, closing, and opening prices of a security.

If the closing price is greater than the opening price, the body (the region between the opening and closing price) is unfilled.

If the opening price is greater than the closing price, the body is filled.

Examples

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

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

plots a candlestick chart with cyan candles.

See Also

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

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

Syntax

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

Arguments

`CouponRate`	Decimal number indicating the annual percentage rate used to determine the coupons payable on a bond.
`Settle`	Settlement date. A vector of serial date numbers or date strings. `Settle` must be earlier than or equal to `Maturity`.
`Maturity`	Maturity date. A vector of serial date numbers or date strings.
`Period`	(Optional) Coupons per year of the bond. A vector of integers. Allowed values are 0, 1, 2 (default), 3, 4, 6, and 12.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. 0 = actual/actual (default), 1 = 30/360 (SIA), 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).
`EndMonthRule`	(Optional) End-of-month rule. A vector. This rule applies only when `Maturity` is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond's coupon payment date is always the same numerical day of the month. 1 = set rule on (default), meaning that a bond's coupon payment date is always the last actual day of the month.
`IssueDate`	(Optional) Date when a bond was issued.
`FirstCouponDate`	(Optional) Date when a bond makes its first coupon payment. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure.

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

Required arguments must be number of bonds (`NUMBONDS`) by 1 or 1-by-`NUMBONDS` conforming vectors or scalars. Optional arguments must be either `NUMBONDS`-by-1 or 1-by-`NUMBONDS` conforming vectors, scalars, or empty matrices.

Description

`[CFlowAmounts, CFlowDates, TFactors, CFlowFlags] = cfamounts(CouponRate, Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate, StartDate, Face)` returns matrices of cash flow amounts, cash flow dates, time factors, and cash flow flags for a portfolio of `NUMBONDS` fixed income securities. The elements contained in the cash flow matrix, time factor matrix, and cash flow flag matrix correspond to the cash flow dates for each security. The first element of each row in the cash flow matrix is the accrued interest payable on each bond. This is zero in the case of all zero coupon bonds. This function determines all cash flows and time mappings for a bond whether or not the coupon structure contains odd first or last periods. All output matrices are padded with `NaN`s as necessary to ensure that all rows have the same number of elements.

`CFlowAmounts` is the cash flow matrix of a portfolio of bonds. Each row represents the cash flow vector of a single bond. Each element in a column represents a specific cash flow for that bond.

`CFlowDates` is the cash flow date matrix of a portfolio of bonds. Each row represents a single bond in the portfolio. Each element in a column represents a cash flow date of that bond.

`TFactors` is the matrix of time factors for a portfolio of bonds. Each row corresponds to the vector of time factors for each bond. Each element in a column corresponds to the specific time factor associated with each cash flow of a bond. Time factors are useful in determining the present value of a stream of cash flows. The term “time factor” refers to the exponent *TF* in the discounting equation

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

where:

PV = present value of a cash flow

CF = the cash flow amount

z = the risk-adjusted annualized rate or yield corresponding to given cash flow. The yield is quoted on a semiannual basis.

TF = time factor for a given cash flow. Time is measured in semiannual periods from the settlement date to the cash flow date. In computing time factors, we use SIA actual/actual day count conventions for all time factor calculations.

`CFlowFlags` is the matrix of cash flow flags for a portfolio of bonds. Each row corresponds to the vector of cash flow flags for each bond. Each element in a column corresponds to the specific flag associated with each cash flow of a bond. Flags identify the type of each cash flow (e.g., nominal coupon cash flow, front or end partial or “stub” coupon, maturity cash flow). Possible values are shown in the table.

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

Flag	Cash Flow Type
2	Larger than normal initial cash flow amount because first coupon period is longer than normal.
3	Nominal coupon cash flow amount.
4	Normal maturity cash flow amount (face value plus the nominal coupon amount).
5	End “stub” coupon amount (last coupon period abnormally short and actual maturity cash flow is smaller than normal).
6	Larger than normal maturity cash flow because last coupon period longer than normal.
7	Maturity cash flow on a coupon bond when the bond has less than one coupon period to maturity.
8	Smaller than normal maturity cash flow when bond has less than one coupon period to maturity.
9	Larger than normal maturity cash flow when bond has less than one coupon period to maturity.
10	Maturity cash flow on a zero coupon bond.

Examples

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

```
Settle = '01-Nov-1993';
Maturity = ['15-Dec-1994';'15-Jun-1995'];
CouponRate= [0.06; 0.05];
Period = [4; 2];
Basis = [1; 0];
[CFlowAmounts, CFlowDates, TFactors, CFlowFlags] = ...
cfamounts(CouponRate,Settle, Maturity, Period, Basis)

CFlowAmounts =

  -0.7667    1.5000    1.5000    1.5000    1.5000  101.5000
```

```
  -1.8989    2.5000    2.5000    2.5000  102.5000       NaN

CFlowDates =

728234     728278     728368     728460     728552     728643
728234     728278     728460     728643     728825        NaN

TFactors =

0    0.2404    0.7403    1.2404    1.7403    2.2404
0    0.2404    1.2404    2.2404    3.2404       NaN

CFlowFlags =

0     3     3     3     3     4
0     3     3     3     4   NaN
```

See Also

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

cfconv

Purpose Cash flow convexity

Syntax `CFlowConvexity = cfconv(CashFlow, Yield)`

Arguments

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

Description `CFlowConvexity = cfconv(CashFlow, Yield)` returns the convexity of a cash flow in periods.

Examples Given a cash flow of nine payments of $2.50 and a final payment $102.50, with a periodic yield of 2.5%

```
CashFlow = [2.5 2.5 2.5 2.5 2.5 2.5 2.5 2.5 2.5 102.5];

Convex = cfconv(CashFlow, 0.025)

Convex =

     90.4493 (periods)
```

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

Purpose

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

Syntax

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

Arguments

`Settle`	Settlement date. A vector of serial date numbers or date strings. `Settle` must be earlier than or equal to `Maturity`.
`Maturity`	Maturity date. A vector of serial date numbers or date strings.
`Period`	(Optional) Coupons per year of the bond. A vector of integers. Allowed values are 0, 1, 2 (default), 3, 4, 6, and 12.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. 0 = actual/actual (default), 1 = 30/360 (SIA), 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).
`EndMonthRule`	(Optional) End-of-month rule. A vector. This rule applies only when `Maturity` is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond's coupon payment date is always the same numerical day of the month. 1 = set rule on (default), meaning that a bond's coupon payment date is always the last actual day of the month.
`IssueDate`	(Optional) Date when a bond was issued.
`FirstCouponDate`	(Optional) Date when a bond makes its first coupon payment. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure.

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

Required arguments must be number of bonds (NUMBONDS) by 1 or 1-by-NUMBONDS conforming vectors or scalars. Optional arguments must be either NUMBONDS-by-1 or 1-by-NUMBONDS conforming vectors, scalars, or empty matrices.

Any input can contain multiple values, but if so, all other inputs must contain the same number of values or a single value that applies to all. For example, if Maturity contains N dates, then Settle must contain N dates or a single date.

Description

CFlowDates = cfdates(Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate, StartDate) returns a matrix of cash flow dates for a bond or set of bonds. cfdates determines all cash flow dates for a bond whether or not the coupon payment structure is normal or the first and/or last coupon period is long or short.

CFlowDates is an N-row matrix of serial date numbers, padded with NaNs as necessary to ensure that all rows have the same number of elements. Use the function datestr to convert serial date numbers to formatted date strings.

Note The cash flow flags for a portfolio of bonds were formerly available as the `cfdates` second output argument, `CFlowFlags`. You can now use `cfamounts` to get these flags. If you specify a `CFlowFlags` argument, `cfdates` displays a message directing you to use `cfamounts`.

Examples

```
CFlowDates = cfdates('14 Mar 1997', '30 Nov 1998', 2, 0, 1)
CFlowDates =
     729541      729724      729906      730089
datestr(CFlowDates)
ans =
31-May-1997
30-Nov-1997
31-May-1998
30-Nov-1998
```

Given three securities with different maturity dates and the same default arguments

```
Maturity = ['30-Sep-1997'; '31-Oct-1998'; '30-Nov-1998'];
CFlowDates = cfdates('14-Mar-1997', Maturity)
CFlowDates =
     729480      729663         NaN         NaN
     729510      729694      729875      730059
     729541      729724      729906      730089
```

Look at the cash-flow dates for the last security.

```
datestr(CFlowDates(3,:))
ans =
31-May-1997
30-Nov-1997
31-May-1998
30-Nov-1998
```

See Also

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

cfdur

Purpose Cash-flow duration and modified duration

Syntax

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

Arguments

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

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

Examples Given a cash flow of nine payments of $2.50 and a final payment $102.50, with a periodic yield of 2.5%

```
CashFlow=[2.5 2.5 2.5 2.5 2.5 2.5 2.5 2.5 2.5 102.5];

[Duration, ModDuration] = cfdur(CashFlow, 0.025)

Duration =
           8.9709 (periods)

ModDuration =
             8.7521 (periods)
```

See Also `bndconvp`, `bndconvy`, `bnddurp`, `bnddury`, `cfconv`

Purpose

Portfolio form of cash flow amounts

Syntax

```
[CFBondDate, AllDates, AllTF, IndByBond] = cfport(CFlowAmounts,
   CFlowDates, TFactors)
```

Arguments

`CFlowAmounts`	Number of bonds (`NUMBONDS`) by number of cash flows (`NUMCFS`) matrix with entries listing cash flow amounts corresponding to each date in `CFlowDates`.
`CFlowDates`	`NUMBONDS`-by-`NUMCFS` matrix with rows listing cash flow dates for each bond and padded with `NaN`s.
`TFactors`	(Optional) `NUMBONDS`-by-`NUMCFS` matrix with entries listing the time between settlement and the cash flow date measured in semiannual coupon periods.

Description

`[CFBondDate, AllDates, AllTF, IndByBond] = cfport(CFlowAmounts, CFlowDates, TFactors)` computes a vector of all cash flow dates of a bond portfolio, and a matrix mapping the cash flows of each bond to those dates. Use the matrix for pricing the bonds against a curve of discount factors.

`CFBondDate` is a `NUMBONDS` by number of dates (`NUMDATES`) matrix of cash flows indexed by bond and by date in `AllDates`. Each row contains a bond's cash flow values at the indices corresponding to entries in `AllDates`. Other indices in the row contain zeros.

`AllDates` is a `NUMDATES`-by-1 list of all dates that have any cash flow from the bond portfolio.

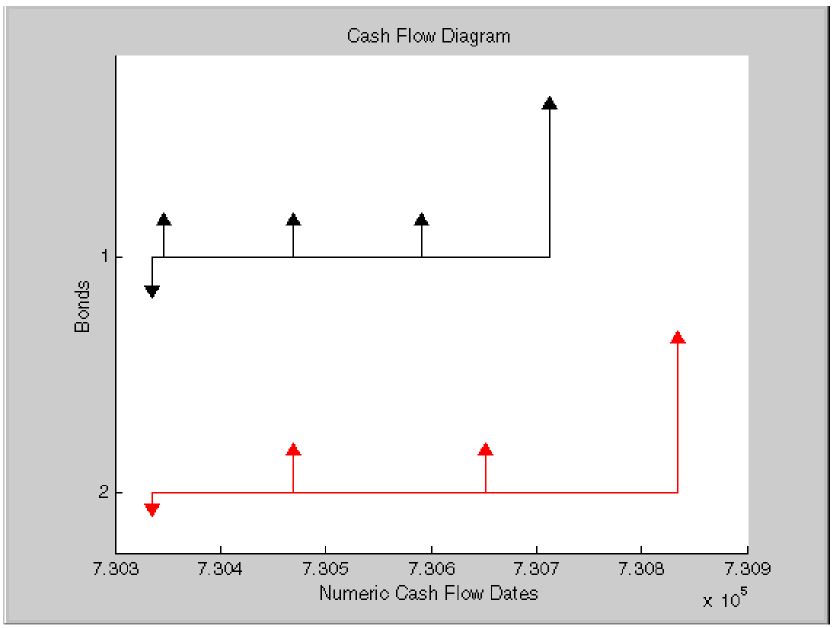
`AllTF` is a `NUMDATES`-by-1 list of time factors corresponding to the dates in `AllDates`. If `TFactors` is not entered, `AllTF` contains the number of days from the first date in `AllDates`.

`IndByBond` is a `NUMBONDS`-by-`NUMCFS` matrix of indices. The *i*th row contains a list of indices into `AllDates` where the *i*th bond has cash flows. Since some bonds have more cash flows than others, the matrix is padded with `NaN`s.

Examples

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

```
Settle = '03-Aug-1999';
Maturity = ['15-Aug-2000';'15-Dec-2000'];
CouponRate= [0.06; 0.05];
Period = [3;2];
Basis = [1;0];
[CFlowAmounts, CFlowDates, TFactors] = cfamounts(CouponRate,...
Settle, Maturity, Period, Basis);
cfplot(CFlowDates,CFlowAmounts)
xlabel('Numeric Cash Flow Dates')
ylabel('Bonds')
title('Cash Flow Diagram')
```



Finally, call `cfport` to map the cash flow amounts to the cash flow dates.

Each row in the resultant `CFBondDate` matrix represents a bond. Each column represents a date on which one or more of the bonds has a cash flow. A 0 means the bond did not have a cash flow on that date. The dates associated with the columns are listed in `AllDates`. For example, the first bond had a cash flow of `2.000` on `730347`. The second bond had no cash flow on this date.

For each bond, `IndByBond` indicates the columns of `CFBondDate`, or dates in `AllDates`, for which a bond has a cash flow.

```
[CFBondDate, AllDates, AllTF, IndByBond] = ...
cfport(CFlowAmounts, CFlowDates, TFactors)

CFBondDate =

  -1.8000  2.0000  2.0000  2.0000       0  102.0000         0
  -0.6694       0  2.5000       0  2.5000         0  102.5000

AllDates =

      730335
      730347
      730469
      730591
      730652
      730713
      730835

AllTF =

         0
    0.0663
    0.7322
    1.3989
    1.7322
    2.0663
    2.7322

IndByBond =

     1     2     3     4     6
     1     3     5     7   NaN
```

See Also

`cfamounts`

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

Syntax

```
TFactors = cftimes(Settle, Maturity, Period, Basis, EndMonthRule,
   IssueDate, FirstCouponDate, LastCouponDate, StartDate)
```

Arguments

`Settle`	Settlement date. A vector of serial date numbers or date strings. `Settle` must be earlier than or equal to `Maturity`.
`Maturity`	Maturity date. A vector of serial date numbers or date strings.
`Period`	(Optional) Coupons per year of the bond. A vector of integers. Allowed values are 0, 1, 2 (default), 3, 4, 6, and 12.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. 0 = actual/actual (default), 1 = 30/360 (SIA), 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).
`EndMonthRule`	(Optional) End-of-month rule. A vector. This rule applies only when `Maturity` is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond's coupon payment date is always the same numerical day of the month. 1 = set rule on (default), meaning that a bond's coupon payment date is always the last actual day of the month.
`IssueDate`	(Optional) Date when a bond was issued.
`FirstCouponDate`	(Optional) Date when a bond makes its first coupon payment. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure.

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

Description

`TFactors = cftimes(Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate, StartDate)` determines the time factors corresponding to the cash flows of a bond or set of bonds. The time factor of a cash flow is the difference between the settlement date and the cash flow date in units of semiannual coupon periods. In computing time factors, we use SIA actual/actual day count conventions for all time factor calculations.

Examples

```
Settle = '15-Mar-1997';
Maturity = '01-Sep-1999';
Period = 2;
TFactors = cftimes(Settle, Maturity, Period)

TFactors =

    0.9239    1.9239    2.9239    3.9239    4.9239
```

See Also

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

corr2cov

Purpose Convert standard deviation and correlation to covariance

Syntax

```
ExpCovariance = corr2cov(ExpSigma, ExpCorrC)
```

Arguments

`ExpSigma`	Vector of length n with the standard deviations of each process. n is the number of random processes.
`ExpCorrC`	(Optional) n-by-n correlation coefficient matrix. If `ExpCorrC` is not specified, the processes are assumed to be uncorrelated, and the identity matrix is used.

Description `corr2cov` converts standard deviation and correlation to covariance.

`ExpCovariance` is an n-by-n covariance matrix, where n is the number of processes.

```
ExpCov(i,j) = ExpCorrC(i,j)*(ExpSigma(i)*ExpSigma(j)
```

Examples

```
ExpSigma = [0.5  2.0];

ExpCorrC = [1.0 -0.5
           -0.5  1.0];

ExpCovariance = corr2cov(ExpSigma, ExpCorrC)
```

Expected results:

```
ExpCovariance =

    0.2500   -0.5000
   -0.5000    4.0000
```

See Also `corrcoef`, `cov`, `cov2corr`, `ewstats`, `std`

Purpose

Convert covariance to standard deviation and correlation coefficient

Syntax

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

Arguments

`ExpCovariance`	n-by-n covariance matrix, e.g., from `cov` or `ewstats`. n is the number of random processes.

Description

`[ExpSigma, ExpCorrC] = cov2corr(ExpCovariance)` converts covariance to standard deviations and correlation coefficients.

`ExpSigma` is a 1-by-n vector with the standard deviation of each process.

`ExpCorrC` is an n-by-n matrix of correlation coefficients.

```
ExpSigma(i) = sqrt(ExpCovariance(i,i))
ExpCorrC(i,j) = ExpCovariance(i,j)/(ExpSigma(i)*ExpSigma(j))
```

Examples

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

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

Expected results:

```
  ExpSigma =

    0.5000    2.0000

ExpCorrC =

    1.0000   -0.5000
   -0.5000    1.0000
```

See Also

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

cpncount

Purpose

Coupon payments remaining until maturity (SIA compliant)

Syntax

```
NumCouponsRemaining = cpncount(Settle, Maturity, Period, Basis,
   EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate,
   StartDate)
```

Arguments

`Settle`	Settlement date. A vector of serial date numbers or date strings. `Settle` must be earlier than or equal to `Maturity`.
`Maturity`	Maturity date. A vector of serial date numbers or date strings.
`Period`	(Optional) Coupons per year of the bond. A vector of integers. Allowed values are 0, 1, 2 (default), 3, 4, 6, and 12.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. 0 = actual/actual (default), 1 = 30/360 (SIA), 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).
`EndMonthRule`	(Optional) End-of-month rule. A vector. This rule applies only when `Maturity` is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond's coupon payment date is always the same numerical day of the month. 1 = set rule on (default), meaning that a bond's coupon payment date is always the last actual day of the month.
`IssueDate`	(Optional) Date when a bond was issued.
`FirstCouponDate`	(Optional) Date when a bond makes its first coupon payment. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure.

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

Required arguments must be number of bonds (`NUMBONDS`) by 1 or 1-by-`NUMBONDS` conforming vectors or scalars. Optional arguments must be either `NUMBONDS`-by-1 or 1-by-`NUMBONDS` conforming vectors, scalars, or empty matrices.

Description

`NumCouponsRemaining = cpncount(Settle, Maturity, Period, Basis, EndMonthRule)` returns the whole number of coupon payments between the settlement and maturity dates for a coupon bond or set of bonds.

Examples

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

n =
     8
```

Given three coupon bonds with different maturity dates and the same default arguments

```
Maturity = ['30 Sep 2000'; '31 Oct 2001'; '30 Nov 2002'];

NumCouponsRemaining = cpncount('14 Sep 1997', Maturity)

NumCouponsRemaining =

     7
     9
    11
```

See Also

accrfrac, cfamounts, cfdates, cftimes, cpndaten, cpndatenq, cpndatep, cpndatepq, cpndaysn, cpndaysp, cpnpersz

Purpose

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

Syntax

```
NextCouponDate = cpndaten(Settle, Maturity, Period, Basis,
   EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate)
```

Arguments

`Settle`	Settlement date. A vector of serial date numbers or date strings. `Settle` must be earlier than or equal to `Maturity`.
`Maturity`	Maturity date. A vector of serial date numbers or date strings.
`Period`	(Optional) Coupons per year of the bond. A vector of integers. Allowed values are 0, 1, 2 (default), 3, 4, 6, and 12.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. 0 = actual/actual (default), 1 = 30/360 (SIA), 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).
`EndMonthRule`	(Optional) End-of-month rule. A vector. This rule applies only when `Maturity` is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond's coupon payment date is always the same numerical day of the month. 1 = set rule on (default), meaning that a bond's coupon payment date is always the last actual day of the month.
`IssueDate`	(Optional) Date when a bond was issued.

`FirstCouponDate`	(Optional) Date when a bond makes its first coupon payment. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure.
`LastCouponDate`	(Optional) Last coupon date of a bond prior to the maturity date. In the absence of a specified `FirstCouponDate`, a specified `LastCouponDate` determines the coupon structure of the bond. The coupon structure of a bond is truncated at the `LastCouponDate` regardless of where it falls and will be followed only by the bond's maturity cash flow date.

Required arguments must be number of bonds (`NUMBONDS`) by 1 or 1-by-`NUMBONDS` conforming vectors or scalars. Optional arguments must be either `NUMBONDS`-by-1 or 1-by-`NUMBONDS` conforming vectors, scalars, or empty matrices.

Description

`NextCouponDate = cpndaten(Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate)` returns the next coupon date after the settlement date. This function finds the next coupon date whether or not the coupon structure is synchronized with the maturity date.

`NextCouponDate` is returned as a serial date number. The function `datestr` converts a serial date number to a formatted date string.

Examples

```
NextCouponDate = cpndaten('14 Mar 1997', '30 Nov 2000', 2, 0, 0);

datestr(NextCouponDate)

ans =

30-May-1997
```

```
NextCouponDate = cpndaten('14 Mar 1997', '30 Nov 2000', 2, 0, 1);

datestr(NextCouponDate)

ans =

31-May-1997

Maturity = ['30 Sep 2000'; '31 Oct 2000'; '30 Nov 2000'];

NextCouponDate = cpndaten('14 Mar 1997', Maturity);

datestr(NextCouponDate)

ans =

31-Mar-1997
30-Apr-1997
31-May-1997
```

See Also

accrfrac, cfamounts, cfdates, cftimes, cpncount, cpndatenq, cpndatep, cpndatepq, cpndaysn, cpndaysp, cpnpersz

cpndatenq

Purpose

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

Syntax

```
NextQuasiCouponDate = cpndatenq(Settle, Maturity, Period, Basis,
   EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate)
```

Arguments

`Settle`	Settlement date. A vector of serial date numbers or date strings. `Settle` must be earlier than or equal to `Maturity`.
`Maturity`	Maturity date. A vector of serial date numbers or date strings.
`Period`	(Optional) Coupons per year of the bond. A vector of integers. Allowed values are 0, 1, 2 (default), 3, 4, 6, and 12.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. 0 = actual/actual (default), 1 = 30/360 (SIA), 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).
`EndMonthRule`	(Optional) End-of-month rule. A vector. This rule applies only when `Maturity` is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond's coupon payment date is always the same numerical day of the month. 1 = set rule on (default), meaning that a bond's coupon payment date is always the last actual day of the month.
`IssueDate`	(Optional) Date when a bond was issued.

`FirstCouponDate`	(Optional) Date when a bond makes its first coupon payment. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure.
`LastCouponDate`	(Optional) Last coupon date of a bond prior to the maturity date. In the absence of a specified `FirstCouponDate`, a specified `LastCouponDate` determines the coupon structure of the bond. The coupon structure of a bond is truncated at the `LastCouponDate` regardless of where it falls and will be followed only by the bond's maturity cash flow date.

Required arguments must be number of bonds (`NUMBONDS`) by 1 or 1-by-`NUMBONDS` conforming vectors or scalars. Optional arguments must be either `NUMBONDS`-by-1 or 1-by-`NUMBONDS` conforming vectors, scalars, or empty matrices. Fill unspecified entries in input vectors with the value `NaN`. Dates can be serial date numbers or date strings.

Description

`NextQuasiCouponDate = cpndatenq(Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate)` determines the next quasi coupon date for a portfolio of `NUMBONDS` fixed income securities whether or not the first or last coupon is normal, short, or long. For zero coupon bonds `cpndatenq` returns quasi coupon dates as if the bond had a semiannual coupon structure. Successive quasi coupon dates determine the length of the standard coupon period for the fixed income security of interest and do not necessarily coincide with actual coupon payment dates.

Outputs are `NUMBONDS`-by-1 vectors.

If `Settle` is a coupon date, this function never returns the settlement date. It returns the quasi coupon date strictly after settlement.

`NextQuasiCouponDate` is returned as a serial date number. The function `datestr` converts a serial date number to a formatted date string.

Examples

Given a pair of bonds with the characteristics

```
Settle = char('30-May-1997','10-Dec-1997');
Maturity = char('30-Nov-2002','10-Jun-2004');
```

Compute `NextCouponDate` for this pair of bonds.

```
NextCouponDate = cpndaten(Settle, Maturity);

datestr(NextCouponDate)

ans =

31-May-1997
10-Jun-1998
```

Compute the next quasi coupon dates for these two bonds.

```
NextQuasiCouponDate = cpndatenq(Settle, Maturity);

datestr(NextQuasiCouponDate)

ans =

31-May-1997
10-Jun-1998
```

Because no `FirstCouponDate` has been specified, the results are identical.

Now supply an explicit `FirstCouponDate` for each bond.

```
FirstCouponDate = char('30-Nov-1997','10-Dec-1998');
```

Compute the next coupon dates.

```
NextCouponDate = cpndaten(Settle, Maturity, 2, 0, 1, [],...
FirstCouponDate);

datestr(NextCouponDate)

ans =

30-Nov-1997
10-Dec-1998
```

The next coupon dates are identical to the specified first coupon dates.

Now recompute the next quasi coupon dates.

```
NextQuasiCouponDate = cpndatenq(Settle, Maturity, 2, 0, 1, [],...
FirstCouponDate);

datestr(NextQuasiCouponDate)

ans =

31-May-1997
10-Jun-1998
```

These results illustrate the distinction between actual coupon payment dates and quasi coupon dates. `FirstCouponDate` (and `LastCouponDate`, as well), when specified, is associated with an actual coupon payment and also serves as the synchronization date for determining all quasi coupon dates. Since each bond in this example pays semiannual coupons, and the first coupon date occurs more than six months after settlement, each will have an intermediate quasi coupon date before the actual first coupon payment occurs.

See Also

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

cpndatep

Purpose

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

Syntax

```
PreviousCouponDate = cpndatep(Settle, Maturity, Period, Basis,
   EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate)
```

Arguments

`Settle`	Settlement date. A vector of serial date numbers or date strings. `Settle` must be earlier than or equal to `Maturity`.
`Maturity`	Maturity date. A vector of serial date numbers or date strings.
`Period`	(Optional) Coupons per year of the bond. A vector of integers. Allowed values are 0, 1, 2 (default), 3, 4, 6, and 12.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. 0 = actual/actual (default), 1 = 30/360 (SIA), 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).
`EndMonthRule`	(Optional) End-of-month rule. A vector. This rule applies only when `Maturity` is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond's coupon payment date is always the same numerical day of the month. 1 = set rule on (default), meaning that a bond's coupon payment date is always the last actual day of the month.
`IssueDate`	(Optional) Date when a bond was issued.

`FirstCouponDate`	(Optional) Date when a bond makes its first coupon payment. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure.
`LastCouponDate`	(Optional) Last coupon date of a bond prior to the maturity date. In the absence of a specified `FirstCouponDate`, a specified `LastCouponDate` determines the coupon structure of the bond. The coupon structure of a bond is truncated at the `LastCouponDate` regardless of where it falls and will be followed only by the bond's maturity cash flow date.

Required arguments must be number of bonds (`NUMBONDS`) by 1 or 1-by-`NUMBONDS` conforming vectors or scalars. Optional arguments must be either `NUMBONDS`-by-1 or 1-by-`NUMBONDS` conforming vectors, scalars, or empty matrices.

Description

`PreviousCouponDate = cpndatep(Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate)` returns the previous coupon date on or before settlement for a portfolio of bonds. This function finds the previous coupon date whether or not the coupon structure is synchronized with the maturity date.

For zero coupon bonds the previous coupon date is the issue date, if available. However, if the issue date is not supplied, the previous coupon date for zero coupon bonds is the previous quasi coupon date calculated as if the frequency is semiannual.

`PreviousCouponDate` is returned as a serial date number. The function `datestr` converts a serial date number to a formatted date string.

Examples

```
PreviousCouponDate = cpndatep('14 Mar 1997', '30 Jun 2000',...
2, 0, 0);

datestr(PreviousCouponDate)

ans =

30-Dec-1996

PreviousCouponDate = cpndatep('14 Mar 1997', '30 Jun 2000',...
2, 0, 1);

datestr(PreviousCouponDate)

ans =

31-Dec-1996

Maturity = ['30 Apr 2000'; '31 May 2000'; '30 Jun 2000'];
PreviousCouponDate = cpndatep('14 Mar 1997', Maturity);

datestr(PreviousCouponDate)

ans =

31-Oct-1996
30-Nov-1996
31-Dec-1996
```

See Also

accrfrac, cfamounts, cfdates, cftimes, cpncount, cpndaten, cpndatenq, cpndatepq, cpndaysn, cpndaysp, cpnpersz

Purpose

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

Syntax

```
PreviousQuasiCouponDate = cpndatepq(Settle, Maturity, Period,
   Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate)
```

Arguments

`Settle`	Settlement date. A vector of serial date numbers or date strings. `Settle` must be earlier than or equal to `Maturity`.
`Maturity`	Maturity date. A vector of serial date numbers or date strings.
`Period`	(Optional) Coupons per year of the bond. A vector of integers. Allowed values are 0, 1, 2 (default), 3, 4, 6, and 12.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. 0 = actual/actual (default), 1 = 30/360 (SIA), 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).
`EndMonthRule`	(Optional) End-of-month rule. A vector. This rule applies only when `Maturity` is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond's coupon payment date is always the same numerical day of the month. 1 = set rule on (default), meaning that a bond's coupon payment date is always the last actual day of the month.
`IssueDate`	(Optional) Date when a bond was issued.

`FirstCouponDate`	(Optional) Date when a bond makes its first coupon payment. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure.
`LastCouponDate`	(Optional) Last coupon date of a bond prior to the maturity date. In the absence of a specified `FirstCouponDate`, a specified `LastCouponDate` determines the coupon structure of the bond. The coupon structure of a bond is truncated at the `LastCouponDate` regardless of where it falls and will be followed only by the bond's maturity cash flow date.

Required arguments must be number of bonds (`NUMBONDS`) by 1 or 1-by-`NUMBONDS` conforming vectors or scalars. Optional arguments must be either `NUMBONDS`-by-1 or 1-by-`NUMBONDS` conforming vectors, scalars, or empty matrices. Fill unspecified entries in input vectors with the value `NaN`. Dates can be serial date numbers or date strings.

Description

```
PreviousQuasiCouponDate = cpndatepq(Settle, Maturity, Period,
Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate)
```

determines the previous quasi coupon date on or before settlement for a set of `NUMBONDS` fixed income securities. This function finds the previous quasi coupon date for a bond with a coupon structure in which the first or last period is either normal, short, or long (whether or not the coupon structure is synchronized to maturity). For zero coupon bonds this function returns quasi coupon dates as if the bond had a semiannual coupon structure.

The term "previous quasi coupon date" refers to the previous coupon date for a bond calculated as if no issue date were specified. Although the issue date is not actually a coupon date, when issue date is specified, the previous actual coupon date for a bond is normally calculated as being either the previous coupon date or the issue date, whichever is greater. This function always returns the previous quasi coupon date regardless of issue date. If the settlement date is a coupon date, this function returns the settlement date.

`PreviousQuasiCouponDate` is returned as a serial date number. The function `datestr` converts a serial date number to a formatted date string.

Examples

Given a pair of bonds with the characteristics

```
Settle = char('30-May-1997','10-Dec-1997');
Maturity = char('30-Nov-2002','10-Jun-2004');
```

With no `FirstCouponDate` explicitly supplied, compute the `PreviousCouponDate` for this pair of bonds.

```
PreviousCouponDate = cpndatep(Settle, Maturity);

datestr(PreviousCouponDate)

ans =

30-Nov-1996
10-Dec-1997
```

Note that since the settlement date for the second bond is also a coupon date, `cpndatep` returns this date as the previous coupon date.

Now establish a `FirstCouponDate` and `IssueDate` for this pair of bonds.

```
FirstCouponDate = char('30-Nov-1997','10-Dec-1998');
IssueDate = char('30-May-1996', '10-Dec-1996');
```

Recompute the `PreviousCouponDate` for this pair of bonds.

```
PreviousCouponDate = cpndatep(Settle, Maturity, 2, 0, 1, ...
IssueDate, FirstCouponDate);

datestr(PreviousCouponDate)

ans =

30-May-1996
10-Dec-1996
```

Since both of these bonds settled before the first coupon had been paid, `cpndatep` returns the `IssueDate` as the `PreviousCouponDate`.

Using the same data, compute `PreviousQuasiCouponDate`.

```
PreviousQuasiCouponDate = cpndatepq(Settle, Maturity, 2, 0, 1,...
IssueDate, FirstCouponDate);

datestr(PreviousQuasiCouponDate)

ans =

30-Nov-1996
10-Dec-1997
```

For the first bond the settlement date is not a normal coupon date. The `PreviousQuasiCouponDate` is the coupon date prior to or on the settlement date. Since the coupon structure is synchronized to `FirstCouponDate`, the previous quasi coupon date is `30-Nov-1996`. `PreviousQuasiCouponDate` disregards `IssueDate` and `FirstCouponDate` in this case. For the second bond the settlement date (`10-Dec-1997`) occurs on a date when a coupon would normally be paid in the absence of an explicit `FirstCouponDate`. `cpndatepq` returns this date as `PreviousQuasiCouponDate`.

See Also

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

Purpose

Number of days to next coupon date (SIA compliant)

Syntax

```
NumDaysNext = cpndaysn(Settle, Maturity, Period, Basis,
   EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate,
   StartDate)
```

Arguments

`Settle`	Settlement date. A vector of serial date numbers or date strings. `Settle` must be earlier than or equal to `Maturity`.
`Maturity`	Maturity date. A vector of serial date numbers or date strings.
`Period`	(Optional) Coupons per year of the bond. A vector of integers. Allowed values are 0, 1, 2 (default), 3, 4, 6, and 12.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. 0 = actual/actual (default), 1 = 30/360 (SIA), 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).
`EndMonthRule`	(Optional) End-of-month rule. A vector. This rule applies only when `Maturity` is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond's coupon payment date is always the same numerical day of the month. 1 = set rule on (default), meaning that a bond's coupon payment date is always the last actual day of the month.
`IssueDate`	(Optional) Date when a bond was issued.
`FirstCouponDate`	(Optional) Date when a bond makes its first coupon payment. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure.

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

Required arguments must be number of bonds (`NUMBONDS`) by `1` or 1-by-`NUMBONDS` conforming vectors or scalars. Optional arguments must be either `NUMBONDS`-by-1 or 1-by-`NUMBONDS` conforming vectors, scalars, or empty matrices.

Description

`NumDaysNext = cpndaysn(Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate, StartDate)` returns the number of days from the settlement date to the next coupon date for a bond or set of bonds. For zero coupon bonds coupon dates are computed as if the bonds have a semiannual coupon structure.

Examples

```
NumDaysNext = cpndaysn('14 Sep 2000', '30 Jun 2001', 2, 0, 0)

NumDaysNext =

   107

NumDaysNext = cpndaysn('14 Sep 2000', '30 Jun 2001', 2, 0, 1)

NumDaysNext =

   108
```

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

NumDaysNext = cpndaysn('14 Sep 2000', Maturity)

NumDaysNext =

    47
    77
   108
```

See Also

accrfrac, cfamounts, cftimes, cfdates, cpncount, cpndaten, cpndatenq, cpndatep, cpndatepq, cpndaysp, cpnpersz

Purpose

Number of days since previous coupon date (SIA compliant)

Syntax

```
NumDaysPrevious = cpndaysp(Settle, Maturity, Period, Basis,
   EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate,
   StartDate)
```

Arguments

`Settle`	Settlement date. A vector of serial date numbers or date strings. `Settle` must be earlier than or equal to `Maturity`.
`Maturity`	Maturity date. A vector of serial date numbers or date strings.
`Period`	(Optional) Coupons per year of the bond. A vector of integers. Allowed values are 0, 1, 2 (default), 3, 4, 6, and 12.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. 0 = actual/actual (default), 1 = 30/360 (SIA), 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).
`EndMonthRule`	(Optional) End-of-month rule. A vector. This rule applies only when `Maturity` is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond's coupon payment date is always the same numerical day of the month. 1 = set rule on (default), meaning that a bond's coupon payment date is always the last actual day of the month.
`IssueDate`	(Optional) Date when a bond was issued.
`FirstCouponDate`	(Optional) Date when a bond makes its first coupon payment. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure.

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

Required arguments must be a number of bonds (`NUMBONDS`) by 1 or 1-by-`NUMBONDS` conforming vectors or scalars. Optional arguments must be either `NUMBONDS`-by-1 or 1-by-`NUMBONDS` conforming vectors, scalars, or empty matrices.

Description

`NumDaysPrevious = cpndaysp(Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate, StartDate)` returns the number of days between the previous coupon date and the settlement date for a bond or set of bonds. When the coupon frequency is 0 (a zero coupon bond), the previous coupon date is calculated as if the frequency were semiannual.

Examples

```
NumDaysPrevious = cpndaysp('14 Mar 2000', '30 Jun 2001', 2, 0, 0)

NumDaysPrevious =

    75

NumDaysPrevious = cpndaysp('14 Mar 2000', '30 Jun 2001', 2, 0, 1)

NumDaysPrevious =

    74
```

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

NumDaysPrevious = cpndaysp('14 Mar 2000', Maturity)

NumDaysPrevious =

   135
   105
    74
```

See Also

accrfrac, cfamounts, cfdates, cftimes, cpncount, cpndaten, cpndatenq, cpndatep, cpndatepq, cpndaysn, cpnpersz

Purpose

Number of days in coupon period (SIA compliant)

Syntax

```
NumDaysPeriod = cpnpersz(Settle, Maturity, Period, Basis,
   EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate,
   StartDate)
```

Arguments

`Settle`	Settlement date. A vector of serial date numbers or date strings. `Settle` must be earlier than or equal to `Maturity`.
`Maturity`	Maturity date. A vector of serial date numbers or date strings.
`Period`	(Optional) Coupons per year of the bond. A vector of integers. Allowed values are 0, 1, 2 (default), 3, 4, 6, and 12.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. 0 = actual/actual (default), 1 = 30/360 (SIA), 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).
`EndMonthRule`	(Optional) End-of-month rule. A vector. This rule applies only when `Maturity` is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond's coupon payment date is always the same numerical day of the month. 1 = set rule on (default), meaning that a bond's coupon payment date is always the last actual day of the month.
`IssueDate`	(Optional) Date when a bond was issued.
`FirstCouponDate`	(Optional) Date when a bond makes its first coupon payment. When `FirstCouponDate` and `LastCouponDate` are both specified, `FirstCouponDate` takes precedence in determining the coupon payment structure.

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

Required arguments must be a number of bonds (`NUMBONDS`) by 1 or 1-by-`NUMBONDS` conforming vectors or scalars. Optional arguments must be either `NUMBONDS`-by-1 or 1-by-`NUMBONDS` conforming vectors, scalars, or empty matrices.

Description

`NumDaysPeriod = cpnpersz(Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate, StartDate)` returns the number of days in the coupon period containing the settlement date. For zero coupon bonds coupon dates are computed as if the bonds have a semiannual coupon structure.

Examples

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

NumDaysPeriod =

   183

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

NumDaysPeriod =

   184
```

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

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

NumDaysPeriod =

   184
   183
   184
```

See Also

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

cur2frac

Purpose Decimal currency values to fractional values

Syntax `Fraction = cur2frac(Decimal, Denominator)`

Description `Fraction = cur2frac(Decimal, Denominator)` converts decimal currency values to fractional values. `Fraction` is returned as a string.

Examples

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

returns `Fraction = 12.1`, a string.

See Also `cur2str`, `frac2cur`

Purpose Bank formatted text

Syntax `String = cur2str(Value, Digits)`

Description `String = cur2str(Value, Digits)` returns the given value in bank format. By default, `Digits = 2`. A negative `Digits` rounds the value to the left of the decimal point. `String` is returned as a string with a leading dollar sign (`$`). Negative numbers are displayed in parentheses.

Examples

```
String = cur2str(–8264, 2)
```

returns `String = ($8264.00)`

See Also `cur2frac`, `frac2cur`

date2time

Purpose Time and frequency from dates

Syntax

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

Arguments

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

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

`Compounding` (Optional) Scalar value representing the rate at which the input zero rates were compounded when annualized. This argument determines the formula for the discount factors:

`Compounding` = `1`, `2`, `3`, `4`, `6`, `12` (Default = `2`.)

`Disc = (1 + Z/F)^(-T)`, where `F` is the compounding frequency, `Z` is the zero rate, and `T` is the time in periodic units, e.g. `T = F` is one year.

`Compounding` = `365`

`Disc = (1 + Z/F)^(-T)`, where `F` is the number of days in the basis year and `T` is a number of days elapsed computed by basis.

`Compounding` = `-1`

`Disc = exp(-T*Z)`, where `T` is time in years.

`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. 0 = actual/actual (default), 1 = 30/360 (SIA), 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).
`EndMonthRule`	(Optional) End-of-month rule. A vector. This rule applies only when `Maturity` is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond's coupon payment date is always the same numerical day of the month. 1 = set rule on (default), meaning that a bond's coupon payment date is always the last actual day of the month.

Description

`[TFactors, F] = date2time(Settle, Dates, Compounding, Basis, EndMonthRule)` computes time factors appropriate to compounded rate quotes beyond the settlement date.

`TFactors` is a vector of time factors.

`F` is a scalar of related compounding frequencies.

`date2time` is the inverse of `time2date`.

See Also

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

dateaxis

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

Syntax

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

Arguments

`Aksis` (Optional) Determines which axis tick labels—*x*, *y*, or *z*—to replace. Enter as a string. Default = `'x'`.

`DateForm` (Optional) Specifies which date format to use. Enter as an integer from 0 to 17. If no `DateForm` argument is entered, this function determines the date format based on the span of the axis limits. For example, if the difference between the axis minimum and maximum is less than 15, the tick labels are converted to three-letter day-of-the-week abbreviations (`DateForm = 8`). See `DateForm` format descriptions below.

`StartDate` (Optional) Assigns the date to the first axis tick value. Enter as a string. The tick values are treated as serial date numbers. The default `StartDate` is the lower axis limit converted to the appropriate date number. For example, a tick value of 1 is converted to the date 01-Jan-0000. Entering `StartDate` as `'06-apr-1999'` assigns the date April 6, 1999 to the first tick value and the axis tick labels are set accordingly.

Description

`dateaxis(Aksis, DateForm, StartDate)` replaces axis tick labels with date labels on a graphic figure.

See the MATLAB `set` command for information on modifying the axis tick values and other axis parameters.

DateForm	Format	Description
0	`01-Mar-1999 15:45:17`	day-month-year hour:minute:second
1	`01-mar-1999`	day-month-year
2	`03/01/99`	month/day/year
3	`Mar`	month, three letters

DateForm	Format	Description
4	M	month, single letter
5	3	month
6	03/01	month/day
7	1	day of month
8	Wed	day of week, three letters
9	W	day of week, single letter
10	1999	year, four digits
11	99	year, two digits
12	Mar99	month year
13	15:45:17	hour:minute:second
14	03:45:17 PM	hour:minute:second AM or PM
15	15:45	hour:minute
16	03:45 PM	hour:minute AM or PM
17	95/03/01	year month day

Examples

```
dateaxis('x') or dateaxis
```

converts the *x*-axis labels to an automatically determined date format.

```
dateaxis('y', 6)
```

converts the *y*-axis labels to the month/day format.

```
dateaxis('x', 2, '03/03/1999')
```

converts the *x*-axis labels to the month/day/year format. The minimum *x*-tick value is treated as March 3, 1999.

See Also

`bolling`, `candle`, `datenum`, `datestr`, `highlow`, `movavg`, `pointfig`

datedisp

Purpose Display date entries

Syntax

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

Arguments

`NumMat`	Numeric matrix to display
`DateForm`	(Optional) Date format. See `datestr` for available and default format flags.

Description

`datedisp(NumMat, DateForm)` displays a matrix with the serial dates formatted as date strings, using a matrix with mixed numeric entries and serial date number entries. Integers between `datenum('01-Jan-1900')` and `datenum('01-Jan-2200')` are assumed to be serial date numbers, while all other values are treated as numeric entries.

`CharMat` is a character array representing `NumMat`. If no output variable is assigned, the function prints the array to the display.

Examples

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

NumMat =

   1.0e+05 *

    7.3073    0.0000    0.0120    7.3010
    7.3073    0.0000    0.0100       NaN

datedisp(NumMat)

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

See Also `datestr`

Purpose Indices of date numbers in matrix

Syntax

```
Indices = datefind(Subset, Superset, Tolerance)
```

Arguments

`Subset`	Subset matrix of date numbers used to find matching date numbers in `Superset`. These date numbers must be a nonrepeating subset of those in `Superset`.
`Superset`	Superset matrix of nonrepeating date numbers whose elements are sought.
`Tolerance`	(Optional) Tolerance (`+/-`) for matching the date numbers in `Superset`. A positive integer. Default = `0`.

Description `Indices = datefind(Subset, Superset, Tolerance)` returns a vector of indices to the date numbers in `Superset` that are present in `Subset`, plus or minus the `Tolerance`. If no date numbers match, `Indices = []`.

Although this function was designed for use with sequential date numbers, you can use it with any nonrepeating integers.

Examples

```
Superset = datenum(1999, 7, 1:31);

Subset = [datenum(1999, 7, 10); datenum(1999, 7, 20)];

Indices = datefind(Subset, Superset, 1)

Indices =

         9
        10
        11
        19
        20
        21
```

See Also `datenum`

datemnth

Purpose

Date of day in future or past month

Syntax

```
TargetDate = datemnth(StartDate, NumberMonths, DayFlag, Basis,
   EndMonthRule)
```

Arguments

`StartDate`	Enter as serial date numbers or date strings.
`NumberMonths`	Vector containing number of months in future (positive) or past (negative). Values must be in integer form.
`DayFlag`	(Optional) Vector containing values that specify how the actual day number for the target date in future or past month is determined. 0 (default) = day number should be the day in the future or past month corresponding to the actual day number of the start date. 1 = day number should be the first day of the future or past month. 2 = day number should be the last day of the future or past month. This flag has no effect if `EndMonthRule` is set to 1.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. 0 = actual/actual (default), 1 = 30/360 (SIA), 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).
`EndMonthRule`	(Optional) End-of-month rule. A vector. 1 = rule in effect, meaning that if you are beginning on the last day of a month, and the month has 30 or fewer days, you will end on the last actual day of the future or past month regardless of whether that month has 28, 29, 30 or 31 days) 0 = rule off (default), meaning that the rule is not in effect.

Any input can contain multiple values, but if so, all other inputs must contain the same number of values or a single value that applies to all. For example, if `StartDate` is an n-row character array of date strings, then `NumberMonths` must be an n-by-1 vector of integers or a single integer. `TargetDate` is then an n-by-1 vector of date numbers.

Description

`TargetDate = datemnth(StartDate, NumberMonths, DayFlag, Basis, EndMonthRule)` returns the serial date number of the target date in the future or past.

Use `datestr` to convert serial date numbers to formatted date strings.

Examples

```
Day = datemnth('3 jun 2001', 6, 0, 0, 0)
Day =
      731188
datestr(Day)
ans =
03-Dec-2001

Day = datemnth('3 jun 2001', 6, 1, 0, 1); datestr(Day)
ans =
01-Dec-2001

Day = datemnth('31 jan 2001', 5, 0, 0, 0); datestr(Day)
ans =
30-Jun-2001

Day = datemnth('31 jan 2001', 5, 1, 0, 0); datestr(Day)
ans =
01-Jun-2001

Day = datemnth('31 jan 2001', 5, 1, 0, 1); datestr(Day)
ans =
30-Jun-2001

Day = datemnth('31 jan 2001', 5, 2, 0, 1); datestr(Day)
ans =
30-Jun-2001

Months = [1; 3; 5; 7; 9];
Day = datemnth('31 jan 2001', Months); datestr(Day)
ans =
28-Feb-2001
30-Apr-2001
30-Jun-2001
31-Aug-2001
```

```
31-Oct-2001
```

See Also `datestr`, `datevec`, `days360`, `days365`, `daysact`, `daysdif`, `wrkdydif`

Purpose Create date number

Syntax

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

Description

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

The date string can be any of several forms.

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

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

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

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

datenum

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

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

Examples

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

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

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

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

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

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

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

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

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

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

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

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

See Also

`datedisp`, `datestr`, `datevec`, `daysact`, `now`, `today`

datestr

Purpose Create date string

Syntax

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

Description

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

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

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

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

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

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

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

Examples

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

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

DateString = datestr(730123, 12)
DateString = Jan99

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

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

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

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

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

See Also

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

Purpose

Date components

Syntax

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

Description

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

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

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

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

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

Examples

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

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

```
DateVec = datevec(730695.776)

DateVec =
        2000      7      28      18      37      26.4

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

Year =
  2000

Month =
     7

Day =
    28

Hour =
    18

Minute =
    37

Second =
    26.4

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

Year =
    2000    2000    2000

Month =
       7       7       7

Day =
      28      29      30
```

See Also datenum, datestr, now, today

Purpose

Date of future or past workday

Syntax

```
EndDate = datewrkdy(StartDate, NumberWorkDays, NumberHolidays)
```

Arguments

`StartDate`	Start date vector. Enter as serial date numbers or date strings.
`NumberWorkDays`	Vector containing number of work or business days in future (positive) or past (negative), including the starting date.
`NumberHolidays`	Vector containing values for the number of holidays within `NumberWorkDays`. `NumberHolidays` and `NumberWorkDays` must have the same sign.

Any input can contain multiple values, but if so, all other inputs must contain the same number of values or a single value that applies to all. For example, if `StartDate` is an n-row character array of date strings, then `NumberWorkDays` must be an n-by-1 vector of integers or a single integer. `EndDate` is then an n-by-1 vector of date numbers.

Description

`EndDate = datewrkdy(StartDate, NumberWorkDays, NumberHolidays)` returns the serial number of the date a given number of workdays before or after the start date.

Use `datestr` to convert serial date numbers to formatted date strings.

Examples

```
Workday = datewrkdy('12-dec-2000', 16, 2);
datestr(Workday)
ans =
04-Jan-2001
NumDays = [16; 20; 44];
Workdays = datewrkdy('12-dec-2000', NumDays, 2);
datestr(Workdays)
ans =
4-Jan-2001
10-Jan-2001
13-Feb-2001
```

See Also

`busdate`, `holidays`, `isbusday`, `wrkdydif`

day

Purpose Day of month

Syntax

```
DayMonth = day(Date)
```

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

Examples

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

or

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

returns `DayMonth = 28`

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

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

Syntax `NumDays = days360(StartDate, EndDate)`

Arguments

`StartDate`	Enter as serial date numbers or date strings.
`EndDate`	Enter as serial date numbers or date strings.

Either input can contain multiple values, but if so, the other must contain the same number of values or a single value that applies to all. For example, if `StartDate` is an n-row character array of date strings, then `EndDate` must be an n-by-1 vector of integers or a single integer. `NumDays` is then an n-by-1 vector of date numbers.

Description `NumDays = days360(StartDate, EndDate)` returns the number of days between `StartDate` and `EndDate` based on a 360-day year (i.e., all months contain 30 days). If `EndDate` is earlier than `StartDate`, `NumDays` is negative.

Examples

```
NumDays = days360('15-jan-2000', '15-mar-2000')

NumDays =

    60

MoreDays = ['15-mar-2000'; '15-apr-2000'; '15-jun-2000'];

NumDays = days360('15-jan-2000', MoreDays)

NumDays =

    60
    90
   150
```

See Also `days365`, `daysact`, `daysdif`, `wrkdydif`, `yearfrac`

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

days360e

Purpose Days between dates based on a 360 day year (European)

Syntax

```
NumDays = days360e(StartDate, EndDate)
```

Arguments

`StartDate`	Row vector, column vector, or scalar value in serial date number or date string format.
`EndDate`	Row vector, column vector, or scalar value in serial date number or date string format.

Either input can contain multiple values, but if so, the other must contain the same number of values or a single value that applies to all.

Description

`NumDays = days360e(StartDate, EndDate)` returns a vector or scalar value representing the number of days between `StartDate` and `EndDate` based on a 360-day year (i.e., all months contain 30 days). If `EndDate` is earlier than `StartDate`, `NumDays` is negative.

This day count convention is used primarily in Europe. Under this convention all months contain 30 days.

Examples

Example 1. Use this convention to find the number of days in the month of January.

```
StartDate = '1-Jan-2002';
EndDate = '1-Feb-2002';
NumDays = days360e(StartDate, EndDate)

NumDays =

    30
```

Example 2. Use this convention to find the number of days in February during a leap year.

```
StartDate = '1-Feb-2000';
EndDate = '1-Mar-2000';
NumDays = days360e(StartDate, EndDate)

NumDays =
```

```
    30
```

Example 3. Use this convention to find the number of days in February of a non- leap year.

```
StartDate = '1-Feb-2002';
EndDate = '1-Mar-2002';

NumDays = days360e(StartDate, EndDate)

NumDays =

    30
```

See Also

days360, days360isda, days360psa

days360isda

Purpose

Days between dates based on a 360 day year (ISDA)

Syntax

```
NumDays = days360isda(StartDate, EndDate)
```

Arguments

`StartDate`	Row vector, column vector, or scalar value in serial date number or date string format.
`EndDate`	Row vector, column vector, or scalar value in serial date number or date string format.

Either input can contain multiple values, but if so, the other must contain the same number of values or a single value that applies to all.

Description

`NumDays = days360isda(StartDate, EndDate)` returns a vector or scalar value representing the number of days between `StartDate` and `EndDate` based on a 360-day year (i.e., all months contain 30 days). If `EndDate` is earlier than `StartDate`, `NumDays` is negative.

Under this convention all months contain 30 days.

Examples

Example 1. Use this convention to find the number of days in the month of January.

```
StartDate = '1-Jan-2002';
EndDate = '1-Feb-2002';
NumDays = days360isda(StartDate, EndDate)

NumDays =

  30
```

Example 2. Use this convention to find the number of days in February during a leap year.

```
StartDate = '1-Feb-2000';
EndDate = '1-Mar-2000';
NumDays = days360isda(StartDate, EndDate)

NumDays =

    30
```

Example 3. Use this convention to find the number of days in February of a non- leap year.

```
StartDate = '1-Feb-2002';
EndDate = '1-Mar-2002';
NumDays = days360isda(StartDate, EndDate)

NumDays =

    30
```

See Also

days360, days360e, days360psa

days360psa

Purpose

Days between dates based on a 360 day year (PSA)

Syntax

```
NumDays = days360psa(StartDate, EndDate)
```

Arguments

`StartDate`	Row vector, column vector, or scalar value in serial date number or date string format.
`EndDate`	Row vector, column vector, or scalar value in serial date number or date string format.

Either input can contain multiple values, but if so, the other must contain the same number of values or a single value that applies to all.

Description

`NumDays = days360psa(StartDate, EndDate)` returns a vector or scalar value representing the number of days between `StartDate` and `EndDate` based on a 360-day year (i.e., all months contain 30 days). If `EndDate` is earlier than `StartDate`, `NumDays` is negative.

Under this payment convention all months contain 30 days. In both leap and non-leap years, if the `StartDate` is the last day of February, this day is considered to be day 30 of the month.

Examples

Example 1. Use this convention to find the number of days in between the last day of February and the first day of March during a leap year.

```
StartDate = '29-Feb-2000';
EndDate = '1-Mar-2000';
NumDays = days360psa(StartDate, EndDate)

NumDays =

     1
```

Example 2. Use this convention to find the number of days in between the last day of February and the first day of March during a non-leap year.

```
StartDate = '28-Feb-2002';
EndDate = '1-Mar-2002';
NumDays = days360psa(StartDate, EndDate)
```

```
NumDays =

     1
```

As expected, the number of days in both cases is the same. The convention always assumes that the last day of February is the 30th day.

See Also

`days360`, `days360e`, `days360isda`

Purpose

Days between dates based on 365-day year

Syntax

`NumDays = days365(StartDate, EndDate)`

Arguments

`StartDate`	Enter as serial date numbers or date strings.
`EndDate`	Enter as serial date numbers or date strings.

Either input can contain multiple values, but if so, the other must contain the same number of values or a single value that applies to all. For example, if `StartDate` is an n-row character array of date strings, then `EndDate` must be an n-by-1 vector of integers or a single integer. `NumDays` is then an n-by-1 vector of date numbers.

Description

`NumDays = days365(StartDate, EndDate)` returns the number of days between dates `StartDate` and `EndDate` based on a 365-day year. (All months contain their actual number of days. February always contains 28 days.) If `EndDate` is earlier than `StartDate`, `NumDays` is negative. Enter dates as serial date numbers or date strings.

Examples

```
NumDays = days365('15-jan-2000', '15-mar-2000')

NumDays =

    59

MoreDays = ['15-mar-2000'; '15-apr-2000'; '15-jun-2000'];

NumDays = days365('15-jan-2000', MoreDays)

NumDays =

    59
    90
   151
```

See Also

`days360`, `daysact`, `daysdif`, `wrkdydif`, `yearfrac`

Purpose

Actual number of days between dates

Syntax

`NumDays = daysact(StartDate, EndDate)`

Arguments

`StartDate`	Enter as serial date numbers or date strings.
`EndDate`	(Optional) Enter as serial date numbers or date strings.

Either input can contain multiple values, but if so, the other must contain the same number of values or a single value that applies to all. For example, if `StartDate` is an n-row character array of date strings, then `EndDate` must be an n-row character array of date strings or a single date. `NumDays` is then an n-by-1 vector of numbers.

Description

`NumDays = daysact(StartDate, EndDate)` returns the actual number of days between two dates. Enter dates as serial date numbers or date strings. `NumDays` is negative if `EndDate` is earlier than `StartDate`.

`NumDays = daysact(StartDate)` returns the actual number of days between the MATLAB base date and `StartDate`. In MATLAB, the base date 1 is 1-Jan-0000 A.D. See `datenum` for a similar function.

Examples

```
NumDays = daysact('7-sep-2002',  '25-dec-2002')
NumDays =
   109

NumDays = daysact('9/7/2002')
NumDays =
      731466

MoreDays = ['09/07/2002'; '10/22/2002'; '11/05/2002'];
NumDays = daysact(MoreDays, '12/25/2002')
NumDays =
   109
    64
    50
```

See Also

`datenum`, `datevec`, `days360`, `days365`, `daysdif`

Purpose

Date away from a starting date for any day-count basis

Syntax

```
NumDays = daysadd(StartDate, NumDays, Basis)
```

Arguments

`StartDate`	Start date. Enter as serial date numbers or date strings.
`NumDays`	Integer number of days from start date. Enter a negative integer for dates before start date.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. 0 = actual/actual (default), 1 = 30/360 (SIA), 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).

Note When using the 30/360 day-count basis, it is not always possible to find the exact date `NumDays` number of days away because of a known discontinuity in the method of counting days. A warning is displayed if this occurs.

Description

`NumDays = daysadd(StartDate, NumDays, Basis)` returns a date `NumDays` number of days away from `StartDate`, using the given day-count basis.

Examples

```
NewDate = daysadd('01-Feb-2004', 31)

NewDate =

      732009

datestr(NewDate)

ans =

03-Mar-2004
```

```
NewDate = daysadd('01-Feb-2004', 31, 1)

NewDate =

      732008

datestr(NewDate)

ans =

02-Mar-2004
```

See Also

`daysdif`

References

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

daysdif

Purpose

Days between dates for any day-count basis

Syntax

```
NumDays = daysdif(StartDate, EndDate, Basis)
```

Arguments

`StartDate`	Enter as serial date numbers or date strings.
`EndDate`	Enter as serial date numbers or date strings.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. 0 = actual/actual (default), 1 = 30/360 (SIA), 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).

Any input argument can contain multiple values, but if so, the other inputs must contain the same number of values or a single value that applies to all. For example, if `StartDate` is an n-row character array of date strings, then `EndDate` must be an n-row character array of date strings or a single date. `NumDays` is then an n-by-1 vector of numbers.

Description

`NumDays = daysdif(StartDate, EndDate, Basis)` returns the number of days between dates `StartDate` and `EndDate` using the given day-count basis. Enter dates as serial date numbers or date strings.

This function is a helper function for the bond pricing and yield functions. It is designed to make the code more readable and to eliminate redundant calls within `if` statements.

Examples

```
NumDays = daysdif('3/1/99', '3/1/00', 1)
NumDays =
   360

MoreDays = ['3/1/2001'; '3/1/2002'; '3/1/2003'];
NumDays = daysdif('3/1/98', MoreDays)
NumDays =
         1096
         1461
         1826
```

See Also

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

References

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

dec2thirtytwo

Purpose

Decimal to thirty-second quotation

Syntax

```
[OutNumber, Fractions] = dec2thirtytwo(InNumber, Accuracy)
```

Arguments

`InNumber`	Input number as a decimal fraction.
`Accuracy`	(Optional) Rounding. Default = 1, round down to nearest thirty second. Other values are 2 (nearest half), 4 (nearest quarter) and 10 (nearest decile).

Description

`[OutNumber, Fractions] = dec2thirtytwo(InNumber, Accuracy)` changes a decimal price quotation for a bond or bond future to a fraction with a denominator of 32.

`OutNumber` is `InNumber` rounded downward to the closest integer.

`Fractions` is the fractional part in units of thirty-second with accuracy as prescribed by the input `Accuracy`.

Examples

Two bonds are quoted with decimal prices of 101.78 and 102.96. Convert these prices to fractions with a denominator of 32.

```
InNumber  = [101.78; 102.96];

[OutNumber, Fractions] = dec2thirtytwo(InNumber)

OutNumber =

   101
   102

Fractions =

    25
    31
```

See Also

`thirtytwo2dec`

Purpose

Fixed declining-balance depreciation schedule

Syntax

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

Arguments

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

Description

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

Examples

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

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

returns

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

See Also

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

depgendb

Purpose General declining-balance depreciation schedule

Syntax

```
Depreciation = depgendb(Cost, Salvage, Life, Factor)
```

Arguments

`Cost`	Cost of the asset.
`Salvage`	Estimated salvage value of the asset.
`Life`	Number of periods over which the asset is depreciated.
`Factor`	Depreciation factor. `Factor` = 2 uses the double-declining-balance method.

Description

`Depreciation = depgendb(Cost, Salvage, Life, Factor)` calculates the declining-balance depreciation for each period.

Examples

A car is purchased for $11,000 and is to be depreciated over five years. The estimated salvage value is $1000. Using the double-declining-balance method, the function calculates the depreciation for each year and returns the remaining depreciable value at the end of the life of the car.

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

returns

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

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

Purpose

Remaining depreciable value

Syntax

```
Value = deprdv(Cost, Salvage, Accum)
```

Arguments

`Cost`	Cost of the asset.
`Salvage`	Salvage value of the asset.
`Accum`	Accumulated depreciation of the asset for prior periods.

Description

`Value = deprdv(Cost, Salvage, Accum)` returns the remaining depreciable value for an asset.

Examples

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

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

Accum =
       7200.00

Value = deprdv(13000, 1000, 7200)

Value =
       4800.00
```

See Also

`depfixdb`, `depgendb`, `depsoyd`, `depstln`

depsoyd

Purpose Sum of years' digits depreciation

Syntax

```
Sum = depsoyd(Cost, Salvage, Life)
```

Arguments

`Cost`	Cost of the asset.
`Salvage`	Salvage value of the asset.
`Life`	Depreciable life of the asset in years.

Description

`Sum = depsoyd(Cost, Salvage, Life)` calculates the depreciation for an asset using the sum of years' digits method. `Sum` is a 1-by-`Life` vector of depreciation values with each element corresponding to a year of the asset's life.

Examples

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

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

returns

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

See Also

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

Purpose

Straight-line depreciation schedule

Syntax

```
Depreciation = depstln(Cost, Salvage, Life)
```

Arguments

`Cost`	Cost of the asset.
`Salvage`	Salvage value of the asset.
`Life`	Depreciable life of the asset in years.

Description

`Depreciation = depstln(Cost, Salvage, Life)` calculates straight-line depreciation for an asset.

Examples

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

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

returns

```
Depreciation =
                 1200
```

See Also

`depfixdb`, `depgendb`, `deprdv`, `depsoyd`

Purpose Zero curve given a discount curve

Syntax

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

Arguments

`DiscRates`	Column vector of discount factors, as decimal fractions. In aggregate, the factors in `DiscRates` constitute a discount curve for the investment horizon represented by `CurveDates`.
`CurveDates`	Column vector of maturity dates (as serial date numbers) that correspond to the discount factors in `DiscRates`.
`Settle`	Serial date number that is the common settlement date for the discount rates in `DiscRates`.
`Compounding`	(Optional) Output compounding. A scalar that sets the compounding frequency per year for annualizing the output zero rates. Allowed values are: 1 annual compounding 2 semiannual compounding (default) 3 compounding three times per year 4 quarterly compounding 6 bimonthly compounding 12 monthly compounding 365 daily compounding -1 continuous compounding
`Basis`	(Optional) Day-count basis for annualizing the output zero rates. 0 = actual/actual (default), 1 = 30/360 (SIA), 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).

Description

`[ZeroRates, CurveDates] = disc2zero(DiscRates, CurveDates, Settle, Compounding, Basis)` returns a zero curve given a discount curve and its maturity dates.

`ZeroRates`	Column vector of decimal fractions. In aggregate, the rates in `ZeroRates` constitute a zero curve for the investment horizon represented by `CurveDates`. The zero rates are the yields to maturity on theoretical zero-coupon bonds.
`CurveDates`	Column vector of maturity dates (as serial date numbers) that correspond to the zero rates. This vector is the same as the input vector `CurveDates`.

Examples

Given discount factors `DiscRates` over a set of maturity dates `CurveDates`, and a settlement date `Settle`

```
DiscRates = [0.9996
             0.9947
             0.9896
             0.9866
             0.9826
             0.9786
             0.9745
             0.9665
             0.9552
             0.9466];

CurveDates = [datenum('06-Nov-2000')
              datenum('11-Dec-2000')
              datenum('15-Jan-2001')
              datenum('05-Feb-2001')
              datenum('04-Mar-2001')
              datenum('02-Apr-2001')
              datenum('30-Apr-2001')
              datenum('25-Jun-2001')
              datenum('04-Sep-2001')
              datenum('12-Nov-2001')];

Settle = datenum('03-Nov-2000');
```

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

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

Execute the function

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

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

```
ZeroRates =
    0.0487
    0.0510
    0.0523
    0.0524
    0.0530
    0.0526
    0.0530
    0.0532
    0.0549
    0.0536

CurveDates =
      730796
      730831
      730866
      730887
      730914
      730943
      730971
      731027
      731098
      731167
```

For readability, `DiscRates` and `ZeroRates` are shown here only to the basis point. However, MATLAB computed them at full precision. If you enter `DiscRates` as shown, `ZeroRates` may differ due to rounding.

See Also

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

Purpose

Bank discount rate of a money market security

Syntax

```
DiscRate = discrate(Settle, Maturity, Face, Price, Basis)
```

Arguments

`Settle`	Enter as serial date number or date string. `Settle` must be earlier than or equal to `Maturity`.
`Maturity`	Enter as serial date number or date string.
`Face`	Redemption (par, face) value.
`Price`	Price of the security.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. 0 = actual/actual (default), 1 = 30/360 (SIA), 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).

Description

`DiscRate = discrate(Settle, Maturity, Face, Price, Basis)` finds the bank discount rate of a security. The bank discount rate normalizes by the face value of the security (e.g., U. S. Treasury Bills) and understates the true yield earned by investors.

Examples

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

returns

```
DiscRate =

    0.0501
```

a discount rate of 5.01%.

See Also

`acrudisc`, `fvdisc`, `prdisc`, `ylddisc`

References

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

ecmnfish

Purpose Fisher information matrix

Syntax

```
Fisher = ecmnfish(Data, Covariance, InvCovariance, MatrixFormat)
```

Arguments

Data	NUMSAMPLES-by-NUMSERIES matrix of observed multivariate normal data
Covariance	NUMSERIES-by-NUMSERIES matrix with covariance estimate of Data
InvCovariance	(Optional) Inverse of covariance matrix: inv(Covariance)
MatrixFormat	(Optional) String that identifies parameters included in the Fisher information matrix. If MatrixFormat = [] or '', the default method full is used. The parameter choices are • full — (Default) Compute full Fisher information matrix. • meanonly — Compute only components of the Fisher information matrix associated with the mean.

Description

Fisher = ecmnfish(Data, Covariance, InvCovariance, MatrixFormat) computes a NUMPARAMS-by-NUMPARAMS Fisher information matrix based on current parameter estimates, where

```
NUMPARAMS = NUMSERIES*(NUMSERIES + 3)/2
```

if MatrixFormat = 'full' and

```
NUMPARAMS = NUMSERIES
```

if MatrixFormat = 'meanonly'.

This routine is very slow for NUMSERIES > 10 or NUMSAMPLES > 1000.

The data matrix has NaNs for missing observations. The multivariate normal model has

```
NUMPARAMS = NUMSERIES + NUMSERIES*(NUMSERIES + 1)/2
```

distinct parameters. Therefore, the full Fisher information matrix is of size NUMPARAMS-by-NUMPARAMS. The first NUMSERIES parameters are estimates for the mean of the data in Mean, and the remaining

NUMSERIES*(NUMSERIES + 1)/2 parameters are estimates for the lower-triangular portion of the covariance of the data in Covariance, in row-major order.

If MatrixFormat = 'meanonly', the number of parameters is reduced to NUMPARAMS = NUMSERIES, where the Fisher information matrix is computed for the mean parameters only. In this format, the routine executes fastest.

This routine expects the inverse of the covariance matrix as an input. If you do not pass in the inverse, the routine computes it. You can obtain an approximation for the lower-bound standard errors of estimation of the parameters from

```
Stderr = (1.0/sqrt(NumSamples)) .* sqrt(diag(inv(Fisher)));
```

Because of missing information, these standard errors may be smaller than the estimated standard errors derived from the expected Hessian matrix. To see the difference, compare with standard errors calculated with ecmnhess.

See Also

ecmnhess, ecmnmle

ecmnhess

Purpose

Hessian of negative log-likelihood function

Syntax

```
Hessian = ecmnhess(Data, Covariance, InvCovariance, MatrixFormat)
```

Arguments

`Data`	`NUMSAMPLES`-by-`NUMSERIES` matrix of observed multivariate normal data
`Covariance`	`NUMSERIES`-by-`NUMSERIES` matrix with covariance estimate of `Data`
`InvCovariance`	(Optional) Inverse of covariance matrix: `inv(Covariance)`
`MatrixFormat`	(Optional) String that identifies parameters included in the Hessian matrix. If `MatrixFormat = []` or `''`, the default method `full` is used. The parameter choices are • `full` — (Default) Compute full Hessian matrix. • `meanonly` — Compute only components of the Hessian matrix associated with the mean.

Description

`Hessian = ecmnhess(Data, Covariance, InvCovariance, MatrixFormat)` computes a `NUMPARAMS` -by-`NUMPARAMS` Hessian matrix of the observed negative log-likelihood function based upon current parameter estimates, where

```
NUMPARAMS = NUMSERIES*(NUMSERIES + 3)/2
```

if `MatrixFormat = 'full'` and

```
NUMPARAMS = NUMSERIES
```

if `MatrixFormat = 'meanonly'`.

This routine is very slow for `NUMSERIES > 10` or `NUMSAMPLES > 1000`.

The data matrix has `NaN`s for missing observations. The multivariate normal model has

```
NUMPARAMS = NUMSERIES + NUMSERIES*(NUMSERIES + 1)/2
```

distinct parameters. Therefore, the full Hessian is a `NUMPARAMS`-by-`NUMPARAMS` matrix.

The first `NUMSERIES` parameters are estimates for the mean of the data in `Mean` and the remaining `NUMSERIES*(NUMSERIES + 1)/2` parameters are estimates for the lower-triangular portion of the covariance of the data in `Covariance`, in row-major order.

If `MatrixFormat = 'meanonly'`, the number of parameters is reduced to `NUMPARAMS = NUMSERIES`, where the Hessian is computed for the mean parameters only. In this format, the routine executes fastest.

This routine expects the inverse of the covariance matrix as an input. If you do not pass in the inverse, the routine computes it.

The equation

```
Stderr = (1.0/sqrt(NumSamples)) .* sqrt(diag(inv(Hessian)));
```

provides an approximation for the observed standard errors of estimation of the parameters.

Because of the additional uncertainties introduced by missing information, these standard errors may be larger than the estimated standard errors derived from the Fisher information matrix. To see the difference, compare with standard errors calculated from `ecmnfish`.

See Also

`ecmnfish`, `ecmnmle`

ecmninit

Purpose Initial mean and covariance

Syntax `[Mean, Covariance] = ecmninit(Data, InitMethod)`

Arguments

`Data`	`NUMSAMPLES`-by-`NUMSERIES` matrix with `NUMSAMPLES` samples of a `NUMSERIES`-dimensional random vector. Missing values are indicated by `NaN`s.
`InitMethod`	(Optional) String that identifies one of three defined initialization methods to compute initial estimates for the mean and covariance of the data. If `InitMethod` = `[]` or `''`, the default method `nanskip` is used. The initialization methods are • `nanskip` — (Default) Skip all records with `NaN`s. • `twostage` — Estimate mean. Fill `NaN`s with the mean. Then estimate the covariance. • `diagonal` — Form a diagonal covariance.

Description `[Mean, Covariance] = ecmninit(Data, InitMethod)` creates initial mean and covariance estimates for the function `ecmnmle`. `Mean` is a `NUMSERIES`-by-1 column vector estimate for the mean of `Data`. `Covariance` is a `NUMSERIES`-by-`NUMSERIES` matrix estimate for the covariance of `Data`.

Algorithm

Model

The general model is

$$Z \sim N(Mean, Covariance)$$

where each row of `Data` is an observation of Z.

Each observation of Z is assumed to be iid (independent identically distributed) multivariate normal, and missing values are assumed to be missing at random (MAR).

Initialization Methods

This routine has three initialization methods that cover most cases, each with its advantages and disadvantages.

nanskip. The `nanskip` method works well with small problems (fewer than 10 series or with monotone missing data patterns). It skips over any records with `NaN`s and estimates initial values from complete-data records only. This initialization method tends to yield fastest convergence of the ECM algorithm. This routine switches to the `twostage` method if it determines that significant numbers of records contain `NaN`.

twostage. The `twostage` method is the best choice for large problems (more than 10 series). It estimates the mean for each series using all available data for each series. It then estimates the covariance matrix with missing values treated as equal to the mean rather than as `NaN`s. This initialization method is quite robust but tends to result in slower convergence of the ECM algorithm.

diagonal. The `diagonal` method is a worst-case approach that deals with problematic data, such as disjoint series and excessive missing data (more than 33% missing data). Of the three initialization methods, this method causes the slowest convergence of the ECM algorithm.

See Also

`ecmnmle`

ecmnmle

Purpose

Mean and covariance of incomplete multivariate normal data

Syntax

```
[Mean, Covariance] = ecmnmle(Data, InitMethod, MaxIterations,
   Tolerance, Mean0, Covar0)
```

Arguments

`Data` — `NUMSAMPLES`-by-`NUMSERIES` matrix with `NUMSAMPLES` samples of a `NUMSERIES`-dimensional random vector. Missing values are indicated by `NaN`s. A sample is also called an *observation* or a *record*.

`InitMethod` — (Optional) String that identifies one of three defined initialization methods to compute initial estimates for the mean and covariance of the data. If `InitMethod` = `[]` or `''`, the default method `nanskip` is used. The initialization methods are

- `nanskip` — (Default) Skip all records with `NaN`s.
- `twostage` — Estimate mean. Fill `NaN`s with mean. Then estimate covariance.
- `diagonal` — Form a diagonal covariance.

Note If you supply `Mean0` and `Covar0`, `InitMethod` is not executed.

`MaxIterations` — (Optional) Maximum number of iterations for the expectation conditional maximization (ECM) algorithm. Default = 50.

`Tolerance` — (Optional) Convergence tolerance for the ECM algorithm (Default = `1.0e-8`.) If `Tolerance` $\leq$ 0, perform maximum iterations specified by `MaxIterations` and do not evaluate the objective function at each step unless in display mode, as described below.

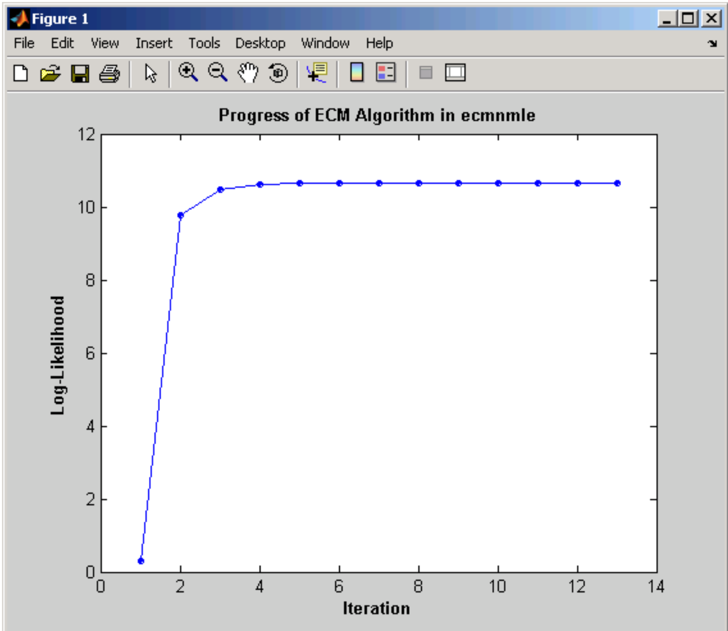
`Mean0`	(Optional) Initial `NUMSERIES`-by-`1` column vector estimate for the mean. If you leave `Mean0` unspecified (`[]`), the method specified by `InitMethod` is used. If you specify `Mean0`, you must also specify `Covar0`.
`Covar0`	(Optional) Initial `NUMSERIES`-by-`NUMSERIES` matrix estimate for the covariance, where the input matrix must be positive-definite. If you leave `Covar0` unspecified (`[]`), the method specified by `InitMethod` is used. If you specify `Covar0`, you must also specify `Mean0`.

Description

`[Mean, Covariance] = ecmnmle(Data, InitMethod, MaxIterations, Tolerance, Mean0, Covar0)` estimates the mean and covariance of a data set. If the data set has missing values, this routine implements the ECM algorithm of Meng and Rubin [2] with enhancements by Sexton and Swensen [3]. ECM stands for *expectation conditional maximization*, a conditional maximization form of the EM algorithm of Dempster, Laird, and Rubin [4].

This routine has two operational modes:

Display Mode. With no output arguments, this mode displays the convergence of the ECM algorithm. It estimates and plots objective function values for each iteration of the ECM algorithm until termination, as shown in the following plot.



Display mode can determine MaxIter and Tolerance values or serve as a diagnostic tool. The objective function is the negative log-likelihood function of the observed data and convergence to a maximum likelihood estimate corresponds with minimization of the objective.

Estimation Mode. With output arguments, this mode estimates the mean and covariance via the ECM algorithm.

Examples

To see an example of how to use ecmnmle, run the demo program ecmguidemo.

Algorithm

Model

The general model is

$$Z \sim N(Mean, Covariance)$$

where each row of Data is an observation of Z.

Each observation of Z is assumed to be iid (independent identically distributed) multivariate normal, and missing values are assumed to be missing at random (MAR). See Little and Rubin [1] for a precise definition of MAR.

This routine estimates the mean and covariance from given data. If data values are missing, the routine implements the ECM algorithm of Meng and Rubin [2] with enhancements by Sexton and Swensen [3].

If a record is empty (every value in a sample is `NaN`), this routine ignores the record because it contributes no information. If such records exist in the data, the number of nonempty samples used in the estimation is ≤ `NumSamples`.

The estimate for the covariance is a biased maximum likelihood estimate (MLE). To convert to an unbiased estimate, multiply the covariance by $Count/Count - 1)$, where *Count* is the number of nonempty samples used in the estimation.

Requirements

This routine requires consistent values for `NUMSAMPLES` and `NUMSERIES` with `NUMSAMPLES` > `NUMSERIES`. It must have enough nonmissing values to converge. Finally, it must have a positive-definite covariance matrix. Although the references provide some necessary and sufficient conditions, general conditions for existence and uniqueness of solutions in the missing-data case do not exist. The main failure mode is an ill-conditioned covariance matrix estimate. Nonetheless, this routine works for most cases that have less than 15% missing data (a typical upper bound for financial data).

Initialization Methods

This routine has three initialization methods that cover most cases, each with its advantages and disadvantages. The ECM algorithm always converges to a minimum of the observed negative log-likelihood function. If you override the initialization methods, you must ensure that the initial estimate for the covariance matrix is positive-definite.

The following is a guide to the supported initialization methods.

nanskip. The `nanskip` method works well with small problems (fewer than 10 series or with monotone missing data patterns). It skips over any records with `NaN`s and estimates initial values from complete-data records only. This initialization method tends to yield fastest convergence of the

ECM algorithm. This routine switches to the `twostage` method if it determines that significant numbers of records contain `NaN`.

twostage. The `twostage` method is the best choice for large problems (more than 10 series). It estimates the mean for each series using all available data for each series. It then estimates the covariance matrix with missing values treated as equal to the mean rather than as `NaN`s. This initialization method is quite robust but tends to result in slower convergence of the ECM algorithm.

diagonal. The `diagonal` method is a worst-case approach that deals with problematic data, such as disjoint series and excessive missing data (more than 33% of data missing). Of the three initialization methods, this method causes the slowest convergence of the ECM algorithm. If problems occur with this method, use display mode to examine convergence and modify either `MaxIterations` or `Tolerance`, or try alternative initial estimates with `Mean0` and `Covar0`. If all else fails, try

```
Mean0 = zeros(NumSeries);
Covar0 = eye(NumSeries,NumSeries);
```

Given estimates for mean and covariance from this routine, you can estimate standard errors with the companion routine `ecmnstd`.

Convergence

The ECM algorithm does not work for all patterns of missing values. Although it works in most cases, it can fail to converge if the covariance becomes singular. If this occurs, plots of the log-likelihood function tend to have a constant upward slope over many iterations as the log of the negative determinant of the covariance goes to zero. In some cases, the objective fails to converge due to machine precision errors. No general theory of missing data patterns exists to determine these cases. An example of a known failure occurs when two time series are proportional wherever both series contain nonmissing values.

See Also

`ecmnfish`, `ecmnhess`, `ecmninit`, `ecmnobj`, `ecmnstd`

References

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

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

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

[4] Dempster, A. P., N. M. Laird, and Donald B. Rubin, "Maximum Likelihood from Incomplete Data via the EM Algorithm," *Journal of the Royal Statistical Society*, Series B, Vol. 39, No. 1, 1977, pp. 1-37.

ecmnobj

Purpose

Multivariate normal negative log-likelihood function

Syntax

```
Objective = ecmnobj(Data, Mean, Covariance, CholCovariance)
```

Arguments

`Data`	`NUMSAMPLES`-by-`NUMSERIES` matrix of observed multivariate normal data
`Mean`	`NUMSERIES`-by-`1` column vector with mean estimate of `Data`
`Covariance`	`NUMSERIES`-by-`NUMSERIES` matrix with covariance estimate of `Data`
`CholCovariance`	(Optional) Cholesky decomposition of covariance matrix: `chol(Covariance)`

Description

`Objective = ecmnobj(Data, Mean, Covariance, CholCovariance)` computes the value of the observed negative log-likelihood function over the data given current estimates for the mean and covariance of the data.

The data matrix has `NaN`s for missing observations. The inputs `Mean` and `Covariance` are current estimates for model parameters.

This routine expects the Cholesky decomposition of the covariance matrix as an input. The routine computes the Cholesky decomposition if you do not explicitly specify it.

See Also

`chol`, `ecmnmle`

Purpose Standard errors for mean and covariance of incomplete data

Syntax

```
[StdMean, StdCovariance] = ecmnstd(Data, Mean, Covariance, Method)
```

Arguments

`Data`	`NUMSAMPLES`-by-`NUMSERIES` matrix with `NUMSAMPLES` samples of a `NUMSERIES`-dimensional random vector. Missing values are indicated by `NaN`s.
`Mean`	`NUMSERIES`-by-`1` column vector of maximum-likelihood parameter estimates for the mean of `Data` using the expectation conditional maximization (ECM) algorithm
`Covariance`	`NUMSERIES`-by-`NUMSERIES` matrix of maximum-likelihood covariance estimates for the covariance of `Data` using the ECM algorithm
`Method`	(Optional) String indicating method of estimation for standard error calculations. The methods are • `hessian` — (Default) Hessian of the observed negative log-likelihood function. • `fisher` — Fisher information matrix.

Description

`[StdMean, StdCovariance] = ecmnstd(Data, Mean, Covariance, Method)` computes standard errors for mean and covariance of incomplete data.

`StdMean` is a `NUMSERIES`-by-`1` column vector of standard errors of estimates for each element of the mean vector `Mean`.

`StdCovariance` is a `NUMSERIES`-by-`NUMSERIES` matrix of standard errors of estimates for each element of the covariance matrix `Covariance`.

Use this routine after estimating the mean and covariance of `Data` with `ecmnmle`. If the mean and distinct covariance elements are treated as the parameter θ in a complete-data maximum-likelihood estimation, then as the number of samples increases, θ attains asymptotic normality such that

$$\theta - E[\theta] \sim N(0, I^{-1}(\theta))$$

where $E[\theta]$ is the mean and $I(\theta)$ is the Fisher information matrix.

With missing data, the Hessian $H(\theta)$ is a good approximation for the Fisher information (which can only be approximated when data is missing).

It is usually advisable to use the default `Method` since the resultant standard errors incorporate the increased uncertainty due to missing data. In particular, standard errors calculated with the Hessian are generally larger than standard errors calculated with the Fisher information matrix.

Note This routine is very slow for `NUMSERIES > 10` or `NUMSAMPLES > 1000`.

See Also

`ecmnmle`

Purpose Effective rate of return

Syntax

```
Return = effrr(Rate, NumPeriods)
```

Arguments

`Rate`	Annual percentage rate. Enter as a decimal fraction.
`NumPeriods`	Number of compounding periods per year, an integer.

Description

`Return = effrr(Rate, NumPeriods)` calculates the annual effective rate of return. Compounding continuously returns `Return` equivalent to `(e^Rate-1)`.

Examples

Find the effective annual rate of return based on an annual percentage rate of 9% compounded monthly.

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

returns

```
Return =

        0.0938
```

or 9.38%

See Also

`nomrr`

eomdate

Purpose Last date of month

Syntax

```
DayMonth = eomdate(Year, Month)
```

Description

`DayMonth = eomdate(Year, Month)` returns the serial date number of the last date of the month for the given year and month. Enter `Year` as a four-digit integer; enter `Month` as an integer from 1 to 12.

Either input argument can contain multiple values, but if so, the other input must contain the same number of values or a single value that applies to all. For example, if `Year` is a 1-by-n vector of integers, then `Month` must be a 1-by-n vector of integers or a single integer. `DayMonth` is then a 1-by-n vector of date numbers.

Use the function `datestr` to convert serial date numbers to formatted date strings.

Examples

```
DayMonth = eomdate(2001, 2)
DayMonth =
      730910
datestr(DayMonth)

ans =
28-Feb-2001

Year = [2002 2003 2004 2005];
DayMonth = eomdate(Year, 2)
DayMonth =
      731275      731640      732006      732371

datestr(DayMonth)

ans =
28-Feb-2002
28-Feb-2003
29-Feb-2004
28-Feb-2005
```

See Also `day`, `eomday`, `lbusdate`, `month`, `year`

Purpose Last day of month

Syntax

```
Day = eomday(Year, Month)
```

Description `Day = eomday(Year, Month)` returns the last day of the month for the given year and month. Enter `Year` as a four-digit integer; enter `Month` as an integer from `1` to `12`.

Either input argument can contain multiple values, but if so, the other input must contain the same number of values or a single value that applies to all. For example, if `Year` is a 1-by-n vector of integers, then `Month` must be a 1-by-n vector of integers or a single integer. `Day` is then a 1-by-n vector of days.

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

Examples

```
Day = eomday(2000, 2)

Day =

    29
```

See Also `day`, `eomdate`, `month`

Purpose

Expected return and covariance from return time series

Syntax

```
[ExpReturn, ExpCovariance, NumEffObs] = ewstats(RetSeries,
   DecayFactor, WindowLength)
```

Arguments

RetSeries	Return Series: number of observations (NUMOBS) by number of assets (NASSETS) matrix of equally spaced incremental return observations. The first row is the oldest observation, and the last row is the most recent.
DecayFactor	(Optional) Controls how much less each observation is weighted than its successor. The *k*th observation back in time has weight DecayFactor^k. DecayFactor must lie in the range: 0 < DecayFactor <= 1. Default = 1, the equally weighted linear moving average model (BIS).
WindowLength	(Optional) Number of recent observations in the computation. Default = NUMOBS.

Description

[ExpReturn, ExpCovariance, NumEffObs] = ewstats(RetSeries, DecayFactor, WindowLength) computes estimated expected returns, estimated covariance matrix, and the number of effective observations.

ExpReturn is a 1-by-NASSETS vector of estimated expected returns.

ExpCovariance is an NASSETS-by-NASSETS estimated covariance matrix. The standard deviations of the asset return processes are given by

```
STDVec = sqrt(diag(ExpCovariance))
```

The correlation matrix is

```
CorrMat = ExpCovariance./( STDVec*STDVec' )
```

NumEffObs is the number of effective observations = (1-DecayFactor^WindowLength)/(1-DecayFactor).

A smaller DecayFactor or WindowLength emphasizes recent data more strongly but uses less of the available data set.

Examples

```
RetSeries = [ 0.24 0.08
              0.15 0.13
              0.27 0.06
              0.14 0.13 ];

DecayFactor = 0.98;

[ExpReturn, ExpCovariance] = ewstats(RetSeries, DecayFactor)

ExpReturn =

    0.1995    0.1002

ExpCovariance =

    0.0032   -0.0017
   -0.0017    0.0010
```

See Also

cov, mean

fbusdate

Purpose First business date of month

Syntax

```
Date = fbusdate(Year, Month, Holiday, Weekend)
```

Arguments

`Year`	Enter as four-digit integer.
`Month`	Enter as integer from 1 to 12.
`Holiday`	(Optional) Vector of holidays and nontrading-day dates. All dates in `Holiday` must be the same format: either serial date numbers or date strings. (Using date numbers improves performance.) The `holidays` function supplies the default vector.
`Weekend`	(Optional) Vector of length 7, containing 0 and 1, the value 1 indicating weekend days. The first element of this vector corresponds to Sunday. Thus, when Saturday and Sunday form the weekend (default), then `Weekend = [1 0 0 0 0 0 1]`.

Description

`Date = fbusdate(Year, Month, Holiday, Weekend)` returns the serial date number for the first business date of the given year and month. `Holiday` specifies nontrading days.

`Year` and `Month` can contain multiple values. If one contains multiple values, the other must contain the same number of values or a single value that applies to all. For example, if `Year` is a 1-by-n vector of integers, then `Month` must be a 1-by-n vector of integers or a single integer. `Date` is then a 1-by-n vector of date numbers.

Use the function `datestr` to convert serial date numbers to formatted date strings.

Examples

Example 1:

```
Date = fbusdate(2001, 11); datestr(Date)
ans =
01-Nov-2001

Year = [2002 2003 2004];
Date = fbusdate(Year, 11); datestr(Date)

ans =
```

```
01-Nov-2002
03-Nov-2003
01-Nov-2004
```

Example 2: You can indicate that Saturday is a business day by appropriately setting the `Weekend` argument.

```
Weekend = [1 0 0 0 0 0 0];
```

March 1, 2003, is a Saturday. Use `fbusdate` to check that this Saturday is actually the first business day of the month.

```
Date = datestr(fbusdate(2003, 3, [], Weekend))

Date =

01-Mar-2003
```

See Also

`busdate`, `eomdate`, `holidays`, `isbusday`, `lbusdate`

frac2cur

Purpose Fractional currency value to decimal value

Syntax `Decimal = frac2cur(Fraction, Denominator)`

Description `Decimal = frac2cur(Fraction, Denominator)` converts a fractional currency value to a decimal value. `Fraction` is the fractional currency value input as a string, and `Denominator` is the denominator of the fraction.

Examples

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

returns

```
Decimal =
         12.1250
```

See Also `cur2frac`, `cur2str`

Purpose Mean-variance efficient frontier

Syntax

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

Arguments

`ExpReturn`	1 by number of assets (`NASSETS`) vector specifying the expected (mean) return of each asset.
`ExpCovariance`	`NASSETS`-by-`NASSETS` matrix specifying the covariance of asset returns.
`NumPorts`	(Optional) Number of portfolios generated along the efficient frontier. Returns are equally spaced between the maximum possible return and the minimum risk point. If `NumPorts` is empty (entered as `[]`, `frontcon` computes 10 equally spaced points. When entering a target rate of return (`PortReturn`), enter `NumPorts` as an empty matrix `[]`.
`PortReturn`	(Optional) Vector of length equal to the number of portfolios (`NPORTS`) containing the target return values on the frontier. If `PortReturn` is not entered or `[]`, `NumPorts` equally spaced returns between the minimum and maximum possible values are used.
`AssetBounds`	(Optional) 2-by-`NASSETS` matrix containing the lower and upper bounds on the weight allocated to each asset in the portfolio. Default lower bound = all 0s (no short-selling). Default upper bound = all 1s (any asset may constitute the entire portfolio).

`Groups`	(Optional) Number of groups (`NGROUPS`)-by-`NASSETS` matrix specifying `NGROUPS` asset groups or classes. Each row specifies a group. `Groups(i,j) = 1` (*j*th asset belongs in the `i`th group). `Groups(i,j) = 0` (*j*th asset not a member of the *i*th group).
`GroupBounds`	(Optional) `NGROUPS`-by-2 matrix specifying, for each group, the lower and upper bounds of the total weights of all assets in that group. Default lower bound = all 0s. Default upper bound = all 1s.

Description

`[PortRisk, PortReturn, PortWts] = frontcon(ExpReturn, ExpCovariance, NumPorts, PortReturn, AssetBounds, Groups, GroupBounds)` returns the mean-variance efficient frontier with user-specified asset constraints, covariance, and returns. For a collection of `NASSETS` risky assets, computes a portfolio of asset investment weights that minimize the risk for given values of the expected return. The portfolio risk is minimized subject to constraints on the asset weights or on groups of asset weights.

`PortRisk` is an `NPORTS`-by-1 vector of the standard deviation of each portfolio.

`PortReturn` is a `NPORTS`-by-1 vector of the expected return of each portfolio.

`PortWts` is an `NPORTS`-by-`NASSETS` matrix of weights allocated to each asset. Each row represents a portfolio. The total of all weights in a portfolio is 1.

`frontcon` generates a plot of the efficient frontier if you invoke it without output arguments.

The asset returns are assumed to be jointly normal, with expected mean returns of `ExpReturn` and return covariance `ExpCovariance`. The variance of a portfolio with 1-by-`NASSETS` weights `PortWts` is given by `PortVar = PortWts*ExpCovariance*PortWts'`. The portfolio expected return is `PortReturn = dot(ExpReturn, PortWts)`.

Examples

Given three assets with expected returns of

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

and expected covariance of

```
ExpCovariance = [ 0.0100   -0.0061    0.0042
                 -0.0061    0.0400   -0.0252
                  0.0042   -0.0252    0.0225];
```

compute the mean-variance efficient frontier for four points.

```
NumPorts = 4;
[PortRisk, PortReturn, PortWts] = frontcon(ExpReturn,...
ExpCovariance, NumPorts)

PortRisk =

    0.0426
    0.0483
    0.1089
    0.2000

PortReturn =

    0.1569
    0.1713
    0.1856
    0.2000

PortWts =

    0.2134    0.3518    0.4348
    0.0096    0.4352    0.5552
         0    0.7128    0.2872
         0    1.0000         0
```

See Also

`ewstats`, `portopt`, `portstats`

fvdisc

Purpose

Future value of discounted security

Syntax

```
FutureVal = fvdisc(Settle, Maturity, Price, Discount, Basis)
```

Arguments

`Settle`	Settlement date. Enter as serial date number or date string. `Settle` must be earlier than or equal to `Maturity`.
`Maturity`	Maturity date. Enter as serial date number or date string.
`Price`	Price (present value) of the security.
`Discount`	Bank discount rate of the security. Enter as decimal fraction.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. 0 = actual/actual (default), 1 = 30/360 (SIA), 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).

Description

`FutureVal = fvdisc(Settle, Maturity, Price, Discount, Basis)` finds the amount received at maturity for a fully vested security.

Examples

Using this data

```
Settle = '02/15/2001';
Maturity = '05/15/2001';
Price = 100;
Discount = 0.0575;
Basis = 2;

FutureVal = fvdisc(Settle, Maturity, Price, Discount, Basis)
```

returns

```
FutureVal =
            101.44
```

See Also

`acrudisc`, `discrate`, `prdisc`, `ylddisc`

References

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

Purpose

Future value with fixed periodic payments

Syntax

```
FutureVal = fvfix(Rate, NumPeriods, Payment, PresentVal, Due)
```

Arguments

`Rate`	Periodic interest rate, as a decimal fraction.
`NumPeriods`	Number of periods.
`Payment`	Periodic payment.
`PresentVal`	(Optional) Initial value. Default = 0.
`Due`	(Optional) When payments are due or made: 0 = end of period (default), or 1 = beginning of period.

Description

`FutureVal = fvfix(Rate, NumPeriods, Payment, PresentVal, Due)` returns the future value of a series of equal payments.

Examples

A savings account has a starting balance of $1500. $200 is added at the end of each month for 10 years and the account pays 9% interest compounded monthly. Using this data

```
FutureVal = fvfix(0.09/12, 12*10, 200, 1500, 0)
```

returns

```
FutureVal =
                42379.89
```

See Also

`fvvar`, `pvfix`, `pvvar`

fvvar

Purpose Future value of varying cash flow

Syntax

```
FutureVal = fvvar(CashFlow, Rate, IrrCFDates)
```

Arguments

`CashFlow`	A vector of varying cash flows. Include the initial investment as the initial cash flow value (a negative number).
`Rate`	Periodic interest rate. Enter as a decimal fraction.
`IrrCFDates`	(Optional) For irregular (nonperiodic) cash flows, a vector of dates on which the cash flows occur. Enter dates as serial date numbers or date strings. Default assumes `CashFlow` contains regular (periodic) cash flows.

Description `FutureVal = fvvar(CashFlow, Rate, IrrCFDates)` returns the future value of a varying cash flow.

Examples This cash flow represents the yearly income from an initial investment of $10,000. The annual interest rate is 8%.

Year 1	$2000
Year 2	$1500
Year 3	$3000
Year 4	$3800
Year 5	$5000

For the future value of this regular (periodic) cash flow

```
FutureVal = fvvar([-10000 2000 1500 3000 3800 5000], 0.08)
```

returns

```
FutureVal =

               2520.47
```

An investment of $10,000 returns this irregular cash flow. The original investment and its date are included. The periodic interest rate is 9%.

Cash flow	Dates
($10000)	January 12, 2000
$2500	February 14, 2001
$2000	March 3, 2001
$3000	June 14, 2001
$4000	December 1, 2001

To calculate the future value of this irregular (nonperiodic) cash flow

```
CashFlow = [-10000, 2500, 2000, 3000, 4000];

IrrCFDates = ['01/12/2000'
              '02/14/2001'
              '03/03/2001'
              '06/14/2001'
              '12/01/2001'];

FutureVal = fvvar(CashFlow, 0.09, IrrCFDates)
```

returns

```
FutureVal =

            170.66
```

See Also

`fvfix`, `irr`, `payuni`, `pvfix`, `pvvar`

fwd2zero

Purpose

Zero curve given a forward curve

Syntax

```
[ZeroRates, CurveDates] = fwd2zero(ForwardRates, CurveDates,
   Settle, Compounding, Basis)
```

Arguments

`ForwardRates`	A number of bonds (`NUMBONDS`) by 1 vector of annualized implied forward rates, as decimal fractions. In aggregate, the rates in `ForwardRates` constitute an implied forward curve for the investment horizon represented by `CurveDates`. The first element pertains to forward rates from the settlement date to the first curve date.
`CurveDates`	A `NUMBONDS-by-1` vector of maturity dates (as serial date numbers) that correspond to the forward rates.
`Settle`	A serial date number that is the common settlement date for the forward rates.
`Compounding`	(Optional) Output compounding. A scalar that sets the compounding frequency per year for annualizing the output zero rates. Allowed values are: 1 annual compounding 2 semiannual compounding (default) 3 compounding three times per year 4 quarterly compounding 6 bimonthly compounding 12 monthly compounding 365 daily compounding -1 continuous compounding
`Basis`	(Optional) Output day-count basis for annualizing the output zero rates. 0 = actual/actual (default), 1 = 30/360 (SIA), 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).

Description

`[ZeroRates, CurveDates] = fwd2zero(ForwardRates, CurveDates, Settle, Compounding, Basis)` returns a zero curve given an implied forward rate curve and its maturity dates.

`ZeroRates`	A `NUMBONDS`-by-1 vector of decimal fractions. In aggregate, the rates in `ZeroRates` constitute a zero curve for the investment horizon represented by `CurveDates`.
`CurveDates`	A `NUMBONDS`-by-1 vector of maturity dates (as serial date numbers) that correspond to the zero rates in `ZeroRates`. This vector is the same as the input vector `CurveDates`.

Examples

Given an implied forward rate curve over a set of maturity dates, a settlement date, and a compounding rate, compute the zero curve.

```
ForwardRates = [0.0469
                0.0519
                0.0549
                0.0535
                0.0558
                0.0508
                0.0560
                0.0545
                0.0615
                0.0486];

CurveDates = [datenum('06-Nov-2000')
              datenum('11-Dec-2000')
              datenum('15-Jan-2001')
              datenum('05-Feb-2001')
              datenum('04-Mar-2001')
              datenum('02-Apr-2001')
              datenum('30-Apr-2001')
              datenum('25-Jun-2001')
              datenum('04-Sep-2001')
              datenum('12-Nov-2001')];

Settle = datenum('03-Nov-2000');
Compounding = 1;
```

Execute the function

```
[ZeroRates, CurveDates] = fwd2zero(ForwardRates, CurveDates,...
Settle, Compounding)
```

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

```
ZeroRates =

    0.0469
    0.0515
    0.0531
    0.0532
    0.0538
    0.0532
    0.0536
    0.0539
    0.0556
    0.0543

CurveDates =

        730796
        730831
        730866
        730887
        730914
        730943
        730971
        731027
        731098
        731167
```

For readability, `ForwardRates` and `ZeroRates` are shown here only to the basis point. However, MATLAB computed them at full precision. If you enter `ForwardRates` as shown, `ZeroRates` may differ due to rounding.

See Also

`zero2fwd` and other functions for Term Structure of Interest Rates

highlow

Purpose High, low, open, close chart

Syntax

```
highlow(High, Low, Close, Open, Color)
Handles = highlow(High, Low, Close, Open, Color)
```

Arguments

`High`	High prices for a security. A column vector.
`Low`	Low prices for a security. A column vector.
`Close`	Closing prices for a security. A column vector.
`Open`	(Optional) Opening prices for a security. A column vector. To specify `Color` when `Open` is unknown, enter `Open` as an empty matrix `[]`.
`Color`	(Optional) Vertical line color. A string. MATLAB supplies a default color if none is specified. The default color differs depending on the background color of the figure window. See `ColorSpec` in the MATLAB documentation for `color` names.

Description

`highlow(High, Low, Close, Open, Color)` plots the high, low, opening, and closing prices of an asset. Plots are vertical lines whose top is the high, bottom is the low, open is a short horizontal tick to the left, and close is a short horizontal tick to the right.

`Handles = highlow(High, Low, Close, Open, Color)` plots the figure and returns the handles of the lines.

Examples

The high, low, and closing prices for an asset are stored in equal-length vectors `AssetHi`, `AssetLo`, and `AssetCl` respectively

```
highlow(AssetHi, AssetLo, AssetCl, [], 'cyan')
```

plots the price data using cyan lines.

See Also

`bolling`, `candle`, `dateaxis`, `movavg`, `pointfig`

holidays

Purpose

Holidays and nontrading days

Syntax

```
Holidays = holidays(StartDate, EndDate)
```

Arguments

`StartDate`	Start date vector. Enter as serial date numbers or date strings.
`EndDate`	End date vector. Enter as serial date numbers or date strings.

Description

`Holidays = holidays(StartDate, EndDate)` returns a vector of serial date numbers corresponding to the holidays and nontrading days between `StartDate` and `EndDate`, inclusive.

`Holidays = holidays` returns a vector of serial date numbers corresponding to all holidays and nontrading days.

As shipped, this function contains all holidays and special nontrading days for the New York Stock Exchange between 1950 and 2050. You can edit the `holidays.m` file to contain your own holidays and nontrading days. By definition, holidays and nontrading days are those that occur on weekdays.

Examples

```
Holidays = holidays('jan 1 2001', 'jun 23 2001')
```

returns

```
Holidays =

       730852
       730901
       730954
       730999
```

which are the serial date numbers for

01-Jan-2001 (New Year's Day)
19-Feb-2001 (President's Day)
13-Apr-2001 (Good Friday)
28-May-2001 (Memorial Day)

See Also

`busdate`, `fbusdate`, `isbusday`, `lbusdate`

Purpose Hour of date or time

Syntax `Hour = hour(Date)`

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

Examples

```
Hour = hour(730473.5584278936)
```

or

```
Hour = hour('19-dec-1999, 13:24:08.17')
```

returns

```
Hour =
      13
```

See Also `datevec`, `minute`, `second`

Purpose Internal rate of return

Syntax

```
Return = irr(CashFlow)
```

Description `Return = irr(CashFlow)` calculates the internal rate of return for a series of periodic cash flows. `CashFlow` is the cash flow vector. The first entry in `CashFlow` is the initial investment. If the initial investment is negative, `irr` generates a unique result only if all subsequent cash flows are positive. If some future cash flows are negative, `irr` generates nonunique solutions (multiple solutions that are each valid).

If the cash flow payments are monthly, multiply the resulting rate of return by 12 for the annual rate of return. This function calculates only positive rates of return; for nonpositive rates of return, `Return = NaN`.

Examples This cash flow represents the yearly income from an initial investment of $100,000:

Year 1	$10,000
Year 2	$20,000
Year 3	$30,000
Year 4	$40,000
Year 5	$50,000

To calculate the internal rate of return on the investment

```
Return = irr([-100000  10000  20000  30000  40000  50000])
```

returns

```
Return =

        0.1201 (12.01%)
```

See Also `effrr`, `mirr`, `nomrr`, `taxedrr`, `xirr`

References Brealey and Myers, *Principles of Corporate Finance*, Chapter 5

Purpose

True for dates that are business days

Syntax

```
Busday = isbusday(Date, Holiday, Weekend)
```

Arguments

`Date`	Date(s) being checked. Enter as a serial date number or date string. `Date` can contain multiple dates, but they must all be in the same format.
`Holiday`	(Optional) Vector of holidays and nontrading-day dates. All dates in `Holiday` must be the same format: either serial date numbers or date strings. (Using date numbers improves performance.) The `holidays` function supplies the default vector.
`Weekend`	(Optional) Vector of length 7, containing 0 and 1, the value 1 indicating weekend days. The first element of this vector corresponds to Sunday. Thus, when Saturday and Sunday form the weekend (default), then `Weekend = [1 0 0 0 0 0 1]`.

Description

`Busday = isbusday(Date, Holiday, Weekend)` returns logical true (`1`) if `Date` is a business day and logical false (`0`) otherwise.

Examples

Example 1:

```
Busday = isbusday('16 jun 2001')

Busday =

         0

Date = ['15 feb 2001'; '16 feb 2001'; '17 feb 2001'];

Busday = isbusday(Date)

Busday =

         1
         1
         0
```

Example 2: Set June 21, 2003 (a Saturday) as a business day.

```
Weekend = [1 0 0 0 0 0 0];

isbusday('June 21, 2003', [], Weekend)

ans =

     1
```

See Also

busdate, fbusdate, holidays, lbusdate

Purpose

Last business date of month

Syntax

```
Date = lbusdate(Year, Month, Holiday, Weekend)
```

Arguments

`Year`	Enter as four-digit integer.
`Month`	Enter as integer from 1 to 12.
`Holiday`	(Optional) Vector of holidays and nontrading-day dates. All dates in `Holiday` must be the same format: either serial date numbers or date strings. (Using date numbers improves performance.) The `holidays` function supplies the default vector.
`Weekend`	(Optional) Vector of length 7, containing 0 and 1, the value 1 indicating weekend days. The first element of this vector corresponds to Sunday. Thus, when Saturday and Sunday form the weekend (default), then `Weekend = [1 0 0 0 0 0 1]`.

Description

`Date = lbusdate(Year, Month, Holiday, Weekend)` returns the serial date number for the last business date of the given year and month. `Holiday` specifies nontrading days.

`Year` and `Month` can contain multiple values. If one contains multiple values, the other must contain the same number of values or a single value that applies to all. For example, if `Year` is a 1-by-n vector of integers, then `Month` must be a 1-by-n vector of integers or a single integer. `Date` is then a 1-by-n vector of date numbers.

Use the function `datestr` to convert serial date numbers to formatted date strings.

Examples

Example 1.

```
Date = lbusdate(2001, 5)

Date =

      731002

datestr(Date)
```

```
ans =

31-May-2001

c
ans =

31-May-2001
31-May-2002
30-May-2003
```

Example 2: You can indicate that Saturday is a business day by appropriately setting the `Weekend` argument.

```
Weekend = [1 0 0 0 0 0 0];
```

May 31, 2003, is a Saturday. Use `lbusdate` to check that this Saturday is actually the last business day of the month.

```
Date = datestr(lbusdate(2003, 5, [], Weekend))

Date =

31-May-2003
```

See Also

`busdate`, `eomdate`, `fbusdate`, `holidays`, `isbusday`

Purpose

Date of last occurrence of weekday in month

Syntax

```
LastDate = lweekdate(Weekday, Year, Month, NextDay)
```

Arguments

`Weekday`	Weekday whose date you seek. Enter as an integer from 1 through 7: 1 Sunday 2 Monday 3 Tuesday 4 Wednesday 5 Thursday 6 Friday 7 Saturday
`Year`	Year. Enter as a four-digit integer.
`Month`	Month. Enter as an integer from 1 through 12.
`NextDay`	(Optional) Weekday that must occur after `Weekday` in the same week. Enter as an integer from 0 through 7, where 0 = ignore (default) and 1 through 7 are as for `Weekday`.

Any input can contain multiple values, but if so, all other inputs must contain the same number of values or a single value that applies to all. For example, if `Year` is a 1-by-n vector of integers, then `Month` must be a 1-by-n vector of integers or a single integer. `LastDate` is then a 1-by-n vector of date numbers.

Description

`LastDate = lweekdate(Weekday, Year, Month, NextDay)` returns the serial date number for the last occurrence of `Weekday` in the given year and month and in a week that also contains `NextDay`.

Use the function `datestr` to convert serial date numbers to formatted date strings.

Examples

To find the last Monday in June 2001

```
LastDate = lweekdate(2, 2001, 6); datestr(LastDate)

ans =

25-Jun-2001
```

To find the last Monday in a week that also contains a Friday in June 2001

```
LastDate = lweekdate(2, 2001, 6, 6); datestr(LastDate)

ans =

25-Jun-2001
```

To find the last Monday in May for 2001, 2002, and 2003

```
Year = [2001:2003];

LastDate = lweekdate(2, Year, 5)

LastDate =

           730999       731363       731727
datestr(LastDate)

ans =

28-May-2001
27-May-2002
26-May-2003
```

See Also

`eomdate`, `lbusdate`, `nweekdate`

Purpose

MATLAB serial date number to Excel serial date number

Syntax

```
DateNum = m2xdate(MATLABDateNumber, Convention)
```

Arguments

`MATLABDateNumber`	A vector or scalar of MATLAB serial date numbers.
`Convention`	(Optional) Excel date system. A vector or scalar. When `Convention = 0` (default), the Excel 1900 date system is in effect. When `Convention = 1`, the Excel 1904 date system in used. In the Excel 1900 date system, the Excel serial date number 1 corresponds to January 1, 1900 A.D. In the Excel 1904 date system, date number 0 is January 1, 1904 A.D.

Vector arguments must have consistent dimensions.

Description

`DateNum = m2xdate(MATLABDateNumber, Convention)` converts MATLAB serial date numbers to Excel serial date numbers. MATLAB date numbers start with 1 = January 1, 0000 A.D., hence there is a difference of 693961 relative to the 1900 date system, or 695422 relative to the 1904 date system. This function is useful with MATLAB Excel Link.

Examples

Given MATLAB date numbers for Christmas 2001 through 2004

```
DateNum = datenum(2001:2004, 12, 25)

DateNum =

      731210      731575      731940      732306
```

convert them to Excel date numbers in the 1904 system

```
ExDate = m2xdate(DateNum, 1)

ExDate =

       35788       36153       36518       36884
```

or the 1900 system

```
ExDate = m2xdate(DateNum)

ExDate =

      37250      37615      37980      38346
```

See Also datenum, datestr, x2mdate

Purpose

Minute of date or time

Syntax

```
Minute = minute(Date)
```

Description

`Minute = minute(Date)` returns the minute given a serial date number or a date string.

Examples

```
Minute = minute(731204.5591223380)
```

or

```
Minute = minute('19-dec-2001, 13:25:08.17')
```

returns

```
Minute =

    25
```

See Also

`datevec`, `hour`, `second`

mirr

Purpose Modified internal rate of return

Syntax

```
Return = mirr(CashFlow, FinRate, Reinvest)
```

Arguments

`CashFlow`	Vector of cash flows. The first entry is the initial investment.
`FinRate`	Finance rate for negative cash flow values. Enter as decimal fraction.
`Reinvest`	Reinvestment rate for positive cash flow values, as a decimal fraction.

Description

`Return = mirr(CashFlow, FinRate, Reinvest)` calculates the modified internal rate of return for a series of periodic cash flows. This function calculates only positive rates of return; for nonpositive rates of return, `Return = 0`.

Examples

This cash flow represents the yearly income from an initial investment of $100,000. The finance rate is 9% and the reinvestment rate is 12%.

Year 1	$20,000
Year 2	($10,000)
Year 3	$30,000
Year 4	$38,000
Year 5	$50,000

To calculate the modified internal rate of return on the investment

```
Return = mirr([-100000 20000 -10000 30000 38000 50000], 0.09,...
0.12)
```

returns

```
Return =
           0.0832 (8.32%)
```

See Also

`annurate`, `effrr`, `irr`, `nomrr`, `pvvar`, `xirr`

References

Brealey and Myers, *Principles of Corporate Finance*, Chapter 5

month

Purpose Month of date

Syntax

```
[MonthNum, MonthString] = month(Date)
```

Description

`[MonthNum, MonthString] = month(Date)` returns the month in numeric and string form given a serial date number or a date string.

Examples

```
[MonthNum, MonthString] = month(730368)
```

or

```
[MonthNum, MonthString] = month('05-Sep-1999')
```

returns

```
MonthNum =

      9

MonthString =

Sep
```

See Also

`datevec`, `day`, `year`

Purpose

Number of whole months between dates

Syntax

```
Months = months(StartDate, EndDate, EndMonthFlag)
```

Arguments

StartDate	Enter as serial date numbers or date strings.
EndDate	Enter as serial date numbers or date strings.
EndMonthFlag	(Optional) end-of-month flag. If StartDate and EndDate are end-of-month dates and EndDate has fewer days than StartDate, EndMonthFlag = 1 (default) treats EndDate as the end of a whole month, while EndMonthFlag = 0 does not.

Description

Months = months(StartDate, EndDate, EndMonthFlag) returns the number of whole months between StartDate and EndDate. If EndDate is earlier than StartDate, Months is negative. Enter dates as serial date numbers or date strings.

Any input argument can contain multiple values, but if so, all other inputs must contain the same number of values or a single value that applies to all. For example, if StartDate is an n-row character array of date strings, then EndDate must be an n-row character array of date strings or a single date. Months is then an n-by-1 vector of numbers.

Examples

```
Months = months('may 31 2000', 'jun 30 2000', 1)
Months =
        1

Months = months('may 31 2000','jun 30 2000', 0)
Months =
        0

Dates = ['mar 31 2002'; 'apr 30 2002'; 'may 31 2002'];
Months = months(Dates, 'jun 30 2002')
Months =
        3
        2
        1
```

See Also

yearfrac

Purpose Leading and lagging moving averages chart

Syntax

```
movavg(Asset, Lead, Lag, Alpha)
[Short, Long] = movavg(Asset, Lead, Lag, Alpha)
```

Arguments

`Asset`	Security data, usually a vector of time-series prices.
`Lead`	Number of samples to use in leading average calculation. A positive integer. `Lead` must be less than or equal to `Lag`.
`Lag`	Number of samples to use in the lagging average calculation. A positive integer.
`Alpha`	(Optional) Control parameter that determines the type of moving averages. 0 = simple moving average (default), 0.5 = square root weighted moving average, 1 = linear moving average, 2 = square weighted moving average, etc. To calculate the exponential moving average, set `Alpha ='e'`.

Description

`movavg(Asset, Lead, lag, Alpha)` plots leading and lagging moving averages.

`[Short, Long] = movavg(Asset, Lead, lag, Alpha)` returns the leading `Short` and lagging `Long` moving average data without plotting it.

Examples

If `Asset` is a vector of stock price data

```
movavg(Asset, 3, 20, 1)
```

plots linear three-sample leading and 20-sample lagging moving averages.

See Also

`bolling`, `candle`, `dateaxis`, `highlow`, `pointfig`

Purpose Nominal rate of return

Syntax

```
Return = nomrr(Rate, NumPeriods)
```

Arguments

`Rate`	Effective annual percentage rate. Enter as a decimal fraction.
`NumPeriods`	Number of compounding periods per year, an integer.

Description `Return = nomrr(Rate, NumPeriods)` calculates the nominal rate of return.

Examples To find the nominal annual rate of return based on an effective annual percentage rate of 9.38% compounded monthly

```
Return = nomrr(0.0938, 12)
```

returns

```
Return =
          0.0900 (9.0%)
```

See Also `effrr`, `irr`, `mirr`, `taxedrr`, `xirr`

Purpose Current date and time

Syntax

```
Datenum = now
```

Description

`Datenum = now` returns the current date and time as a serial date number.

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

Examples

```
Datenum = now

Datenum =

    730695.5942469908
```

(on July 28, 2000 at 2:15 PM)

See Also `date`, `datenum`, `today`

Purpose

Date of specific occurrence of weekday in month

Syntax

```
Date = nweekdate(n, Weekday, Year, Month, Same)
```

Arguments

`n`	Nth occurrence of the weekday in a month. Enter as integer from 1 through 5.
`Weekday`	Weekday whose date you seek. Enter as integer from 1 through 7. 1 Sunday 2 Monday 3 Tuesday 4 Wednesday 5 Thursday 6 Friday 7 Saturday
`Year`	Year. Enter as a four-digit integer.
`Month`	Month. Enter as an integer from 1 through 12.
`Same`	(Optional) Weekday that must occur in the same week with `Weekday`. Enter as an integer from 0 through 7, where 0 = ignore (default) and 1 through 7 are as for `Weekday`.

Description

`Date = nweekdate(n, Weekday, Year, Month, Same)` returns the serial date number for the specific occurrence of the weekday in the given year and month, and in a week that also contains the weekday `Same`.

If `n` is larger than the last occurrence of `Weekday`, `Date = 0`.

Any input can contain multiple values, but if so, all other inputs must contain the same number of values or a single value that applies to all. For example, if `Year` is a 1-by-n vector of integers, then `Month` must be a 1-by-n vector of integers or a single integer. `Date` is then a 1-by-n vector of date numbers.

Use the function `datestr` to convert serial date numbers to formatted date strings.

Examples

To find the first Thursday in May 2001

```
Date = nweekdate(1, 5, 2001, 5); datestr(Date)

ans =

03-May-2001
```

To find the first Thursday in a week that also contains a Wednesday in May 2001

```
Date = nweekdate(2, 5, 2001, 5, 4); datestr(Date)

ans =

10-May-2001
```

To find the third Monday in February for 2001, 2002, and 2003

```
Year = [2001:2003];

Date = nweekdate(3, 2, Year, 2)

Date =
       730901      731265      731629

datestr(Date)

ans =

19-Feb-2001
18-Feb-2002
17-Feb-2003
```

See Also

`fbusdate`, `lbusdate`, `lweekdate`

Purpose Option profit

Syntax

```
Profit = opprofit(AssetPrice, Strike, Cost, PosFlag, OptType)
```

Arguments

`AssetPrice`	Asset price.
`Strike`	Strike or exercise price.
`Cost`	Cost of the option.
`PosFlag`	Option position. 0 = long, 1 = short.
`OptType`	Option type. 0 = call option, 1 = put option.

Description

`Profit = opprofit(AssetPrice, Strike, Cost, PosFlag, OptType)` returns the profit of an option.

Examples

Buying (going long on) a call option with a strike price of $90 on an underlying asset with a current price of $100 for a cost of $4

```
Profit = opprofit(100, 90, 4, 0, 0)
```

returns

```
Profit =
          6.00
```

a profit of $6 if the option is exercised under these conditions.

See Also

`binprice`, `blsprice`

payadv

Purpose Periodic payment given number of advance payments

Syntax

```
Payment = payadv(Rate, NumPeriods, PresentValue, FutureValue,
  Advance)
```

Arguments

`Rate`	Lending or borrowing rate per period. Enter as a decimal fraction. Must be greater than or equal to 0.
`NumPeriods`	Number of periods in the life of the instrument.
`PresentValue`	Present value of the instrument.
`FutureValue`	Future value or target value to be attained after `NumPeriods` periods.
`Advance`	Number of advance payments. If the payments are made at the beginning of the period, add 1 to `Advance`.

Description

`Payment = payadv(Rate, NumPeriods, PresentValue, FutureValue, Advance)` returns the periodic payment given a number of advance payments.

Examples

The present value of a loan is $1000.00 and it will be paid in full in 12 months. The annual interest rate is 10% and three payments are made at closing time. Using this data

```
Payment = payadv(0.1/12, 12, 1000, 0, 3)
```

returns

```
Payment =

          85.94
```

for the periodic payment.

See Also

`amortize`, `payodd`, `payper`

Purpose

Payment of loan or annuity with odd first period

Syntax

```
Payment = payodd(Rate, NumPeriods, PresentValue, FutureValue, Days)
```

Arguments

`rate`	Interest rate per period. Enter as a decimal fraction.
`NumPeriods`	Number of periods in the life of the instrument.
`PresentValue`	Present value of the instrument.
`FutureValue`	Future value or target value to be attained after `NumPeriods` periods.
`Days`	Actual number of days until the first payment is made.

Description

`Payment = payodd(Rate, NumPeriods, PresentValue, FutureValue, Days)` returns the payment for a loan or annuity with an odd first period.

Examples

A two-year loan for $4000 has an annual interest rate of 11%. The first payment will be made in 36 days. To find the monthly payment

```
Payment = payodd(0.11/12, 24, 4000, 0, 36)
```

returns

```
Payment =

          186.77
```

See Also

`amortize`, `payadv`, `payper`

payper

Purpose Periodic payment of loan or annuity

Syntax

```
Payment = payper(Rate, NumPeriods, PresentValue, FutureValue, Due)
```

Arguments

`Rate`	Interest rate per period. Enter as a decimal fraction.
`NumPeriods`	Number of payment periods in the life of the instrument.
`PresentValue`	Present value of the instrument.
`FutureValue`	(Optional) Future value or target value to be attained after `NumPeriods` periods. Default = 0.
`Due`	(Optional) When payments are due: 0 = end of period (default), or 1 = beginning of period.

Description `Payment = payper(Rate, NumPeriods, PresentValue, FutureValue, Due)` returns the periodic payment of a loan or annuity.

Examples Find the monthly payment for a three-year loan of $9000 with an annual interest rate of 11.75%

```
Payment = payper(0.1175/12, 36, 9000, 0, 0)
```

returns

```
Payment =

          297.86
```

See Also `amortize`, `fvfix`, `payadv`, `payodd`, `pvfix`

Purpose Uniform payment equal to varying cash flow

Syntax

```
Series = payuni(CashFlow, Rate)
```

Arguments

`CashFlow`	A vector of varying cash flows. Include the initial investment as the initial cash flow value (a negative number).
`Rate`	Periodic interest rate. Enter as a decimal fraction.

Description `Series = payuni(CashFlow, Rate)` returns the uniform series value of a varying cash flow.

Examples This cash flow represents the yearly income from an initial investment of $10,000. The annual interest rate is 8%.

Year 1	$2000
Year 2	$1500
Year 3	$3000
Year 4	$3800
Year 5	$5000

To calculate the uniform series value

```
Series = payuni([-10000 2000 1500 3000 3800 5000], 0.08)
```

returns

```
Series =

            429.63
```

See Also `fvfix`, `fvvar`, `irr`, `pvfix`, `pvvar`

pcalims

Purpose Linear inequalities for individual asset allocation

Syntax

```
[A,b] = pcalims(AssetMin, AssetMax, NumAssets)
```

Arguments

AssetMin	Scalar or NASSETS vector of minimum allocations in each asset. NaN indicates no constraint.
AssetMax	Scalar or NASSETS vector of maximum allocations in each asset. NaN indicates no constraint.
NumAssets	(Optional) Number of assets. Default = length of AssetMin or AssetMax.

Description

[A,b] = pcalims(AssetMin, AssetMax, NumAssets) specifies the lower and upper bounds of portfolio allocations in each of NumAssets available asset investments.

A is a matrix and b a vector such that A*PortWts' <= b, where PortWts is a 1-by-NASSETS vector of asset allocations.

If pcalims is called with fewer than two output arguments, the function returns A concatenated with b [A,b].

Examples

Set the minimum weight in every asset to 0 (no short-selling), and set the maximum weight of IBM to 0.5 and CSCO to 0.8, while letting the maximum weight in INTC float.

Asset	IBM	INTC	CSCO
Min. Wt.	0	0	0
Max. Wt.	0.5		0.8

```
AssetMin = 0
AssetMax = [0.5 NaN 0.8]
[A,b] = pcalims(AssetMin, AssetMax)

A =
     1     0     0
     0     0     1
    -1     0     0
     0    -1     0
     0     0    -1

b =

    0.5000
    0.8000
         0
         0
         0
```

Portfolio weights of 50% in IBM and 50% in INTC satisfy the constraints.

Set the minimum weight in every asset to 0 and the maximum weight to 1.

Asset	IBM	INTC	CSCO
Min. Wt.	0	0	0
Max. Wt.	1	1	1

```
AssetMin = 0
AssetMax = 1
NumAssets = 3
```

```
[A,b] = pcalims(AssetMin, AssetMax, NumAssets)

A =

      1      0      0
      0      1      0
      0      0      1
     -1      0      0
      0     -1      0
      0      0     -1

b =
     1
     1
     1
     0
     0
     0
```

Portfolio weights of 50% in IBM and 50% in INTC satisfy the constraints.

See Also

`pcgcomp`, `pcglims`, `pcpval`, `portcons`, `portopt`

Purpose

Linear inequalities for asset group comparison constraints

Syntax

```
[A,b] = pcgcomp(GroupA, AtoBmin, AtoBmax, GroupB)
```

Arguments

`GroupA` `GroupB`	Number of groups (`NGROUPS`) by number of assets (`NASSETS`) specifications of groups to compare. Each row specifies a group. For a specific group, `Group(i,j) = 1` if the group contains asset `j`; otherwise, `Group(i,j) = 0`.
`AtoBmin` `AtoBmax`	Scalar or `NGROUPS`-long vectors of minimum and maximum ratios of allocations in `GroupA` to allocations in `GroupB`. `NaN` indicates no constraint between the two groups. Scalar bounds are applied to all group pairs. The total number of assets allocated to `GroupA` divided by the total number of assets allocated to `GroupB` is >= `AtoBmin` and <= `AtoBmax`.

Description

`[A,b] = pcgcomp(GroupA, AtoBmin, AtoBmax, GroupB)` specifies that the ratio of allocations in one group to allocations in another group is at least `AtoBmin` to 1 and at most `AtoBmax` to 1. Comparisons can be made between an arbitrary number of group pairs `NGROUPS` comprising subsets of `NASSETS` available investments.

`A` is a matrix and `b` a vector such that `A*PortWts' <= b`, where `PortWts` is a 1-by-`NASSETS` vector of asset allocations.

If `pcgcomp` is called with fewer than two output arguments, the function returns `A` concatenated with `b` `[A,b]`.

Examples

Asset	INTC	XOM	RD
Region	North America	North America	Europe
Sector	Technology	Energy	Energy

Group	Min. Exposure	Max. Exposure
North America	0.30	0.75
Europe	0.10	0.55
Technology	0.20	0.50
Energy	0.20	0.80

Make the North American energy sector compose exactly 20% of the North American investment.

```
%           INTC  XOM  RD
GroupA = [   0    1    0  ];  % North American Energy
GroupB = [   1    1    0  ];  % North America

AtoBmin = 0.20;
AtoBmax = 0.20;

[A,b] = pcgcomp(GroupA, AtoBmin, AtoBmax, GroupB)

A =

    0.2000    -0.8000     0
   -0.2000     0.8000     0

b =

   0
   0
```

Portfolio weights of 40% for INTC, 10% for XOM, and 50% for RD satisfy the constraints.

See Also

`pcalims`, `pcglims`, `pcpval`, `portcons`, `portopt`

Purpose

Linear inequalities for asset group minimum and maximum allocation

Syntax

```
[A,b] = pcglims(Groups, GroupMin, GroupMax)
```

Arguments

`Groups`	Number of groups (`NGROUPS`) by number of assets (`NASSETS`) specification of which assets belong to which group. Each row specifies a group. For a specific group, `Group(i,j) = 1` if the group contains asset `j`; otherwise, `Group(i,j) = 0`.
`GroupMin` `GroupMax`	Scalar or `NGROUPS`-long vectors of minimum and maximum combined allocations in each group. `NaN` indicates no constraint. Scalar bounds are applied to all groups.

Description

`[A,b] = pcglims(Groups, GroupMin, GroupMax)` specifies minimum and maximum allocations to groups of assets. An arbitrary number of groups, `NGROUPS`, comprising subsets of `NASSETS` investments, is allowed.

`A` is a matrix and `b` a vector such that `A*PortWts' <= b`, where `PortWts` is a 1-by-`NASSETS` vector of asset allocations.

If `pcglims` is called with fewer than two output arguments, the function returns `A` concatenated with `b` `[A,b]`.

Examples

Asset	INTC	XOM	RD
Region	North America	North America	Europe
Sector	Technology	Energy	Energy

Group	**Min. Exposure**	**Max. Exposure**
North America	0.30	0.75
Europe	0.10	0.55
Technology	0.20	0.50
Energy	0.50	0.50

Set the minimum and maximum investment in various groups.

```
%           INTC  XOM  RD
Groups = [   1     1    0  ;  % North America
             0     0    1  ;  % Europe
             1     0    0  ;  % Technology
             0     1    1  ]; % Energy

GroupMin = [0.30
            0.10
            0.20
            0.50];

GroupMax = [0.75
            0.55
            0.50
            0.50];

[A,b] = pcglims(Groups, GroupMin, GroupMax)

A =

    -1    -1     0
     0     0    -1
    -1     0     0
     0    -1    -1
     1     1     0
     0     0     1
     1     0     0
     0     1     1
```

```
b =

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

Portfolio weights of 50% in INTC, 25% in XOM, and 25% in RD satisfy the constraints.

See Also

`pcalims`, `pcgcomp`, `pcpval`, `portcons`, `portopt`

pcpval

Purpose Linear inequalities for fixing total portfolio value

Syntax

```
[A,b] = pcpval(PortValue, NumAssets)
```

Arguments

PortValue	Scalar total value of asset portfolio (sum of the allocations in all assets). PortValue = 1 specifies weights as fractions of the portfolio and return and risk numbers as rates instead of value.
NumAssets	Number of available asset investments.

Description

[A,b] = pcpval(PortValue, NumAssets) scales the total value of a portfolio of NumAssets assets to PortValue. All portfolio weights, bounds, return, and risk values except ExpReturn and ExpCovariance (see portopt) are in terms of PortValue.

A is a matrix and b a vector such that A*PortWts' <= b, where PortWts is a 1-by-NASSETS vector of asset allocations.

If pcpval is called with fewer than two output arguments, the function returns A concatenated with b [A,b].

Examples

Scale the value of a portfolio of three assets to 1, so all return values are rates and all weight values are in fractions of the portfolio.

```
PortValue = 1;
NumAssets = 3;

[A,b] = pcpval(PortValue, NumAssets)

A =

     1     1     1
    -1    -1    -1

b =

    1
   -1
```

Portfolio weights of 40%, 10%, and 50% in the three assets satisfy the constraints.

See Also

`pcalims`, `pcgcomp`, `pcglims`, `portcons`, `portopt`

pointfig

Purpose Point and figure chart

Syntax `pointfig(Asset)`

Description `pointfig(Asset)` plots a point and figure chart for a vector of price data `Asset`. Upward price movements are plotted as X's and downward price movements are plotted as O's.

See Also `bolling`, `candle`, `dateaxis`, `highlow`, `movavg`

Purpose Optimal capital allocation to efficient frontier portfolios

Syntax

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

Arguments

`PortRisk`	Standard deviation of each risky asset efficient frontier portfolio. A number of portfolios (`NPORTS`) by 1 vector.
`PortReturn`	Expected return of each risky asset efficient frontier portfolio. An `NPORTS-by-1` vector.
`PortWts`	Weights allocated to each asset. An `NPORTS` by number of assets (`NASSETS`) matrix of weights allocated to each asset. Each row represents an efficient frontier portfolio of risky assets. Total of all weights in a portfolio is 1.
`RisklessRate`	Risk-free lending rate. A decimal number.
`BorrowRate`	(Optional) Borrowing rate. A decimal number. If borrowing is not desired, or not an option, set to `NaN` (default).
`RiskAversion`	(Optional) Coefficient of investor's degree of risk aversion. Higher numbers indicate greater risk aversion. Typical coefficients range between 2.0 and 4.0 (Default = 3).

Description

`[RiskyRisk, RiskyReturn, RiskyWts, RiskyFraction, OverallRisk, OverallReturn] = portalloc(PortRisk, PortReturn, PortWts, RisklessRate, BorrowRate, RiskAversion)` computes the optimal risky portfolio, and the optimal allocation of funds between the risky portfolio and the risk-free asset.

`RiskyRisk` is the standard deviation of the optimal risky portfolio.

`RiskyReturn` is the expected return of the optimal risky portfolio.

`RiskyWts` is a 1-by-`NASSETS` vector of weights allocated to the optimal risky portfolio. The total of all weights in the portfolio is 1.

`RiskyFraction` is the fraction of the complete portfolio allocated to the risky portfolio.

`OverallRisk` is the standard deviation of the optimal overall portfolio.

`OverallReturn` is the expected rate of return of the optimal overall portfolio.

`portalloc` generates a plot of the optimal capital allocation if you invoke it without output arguments.

Examples

Generate the efficient frontier from the asset data.

```
ExpReturn = [0.1 0.2 0.15];

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

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

Find the optimal risky portfolio and allocate capital. The risk free investment return is 8%, and the borrowing rate is 12%.

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

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

RiskyRisk =

    0.1283

RiskyReturn =

    0.1788

RiskyWts =

    0.0265    0.6023    0.3712

RiskyFraction =

    1.1898
```

```
OverallRisk =

    0.1527

OverallReturn =

    0.1899
```

See Also `frontcon`, `portrand`, `portstats`

References Bodie, Kane, and Marcus, *Investments*, Second Edition, Chapters 6 and 7.

portcons

Purpose

Portfolio constraints

Syntax

```
ConSet = portcons(varargin)
```

Description

Using linear inequalities, `portcons` generates a matrix of constraints for a portfolio of asset investments. The matrix `ConSet` is defined as `ConSet = [A b]`. `A` is a matrix and `b` a vector such that `A*PortWts' <= b` sets the value, where `PortWts` is a 1 by number of assets (`NASSETS`) vector of asset allocations.

`ConSet = portcons('ConstType', Data1, ..., DataN)` creates a matrix `ConSet`, based on the constraint type `ConstType`, and the constraint parameters `Data1, ..., DataN`.

`ConSet = portcons('ConstType1', Data11, ..., Data1N,'ConstType2', Data21, ..., Data2N, ...)` creates a matrix `ConSet`, based on the constraint types `ConstTypeN`, and the corresponding constraint parameters `DataN1, ..., DataNN`.

Constraint Type	Description	Values
Default	All allocations are >= 0; no short selling allowed. Combined value of portfolio allocations normalized to 1.	`NumAssets` (required). Scalar representing number of assets in portfolio.
`PortValue`	Fix total value of portfolio to `PVal`.	`PVal` (required). Scalar representing total value of portfolio. `NumAssets` (required). Scalar representing number of assets in portfolio. See `pcpval`.

Constraint Type	Description	Values
AssetLims	Minimum and maximum allocation per asset.	AssetMin (required). Scalar or vector of length NASSETS, specifying minimum allocation per asset. AssetMax (required). Scalar or vector of length NASSETS, specifying maximum allocation per asset. NumAssets (optional). See pcalims.
GroupLims	Minimum and maximum allocations to asset group.	Groups (required). NGROUPS-by-NASSETS matrix specifying which assets belong to each group. GroupMin (required). Scalar or a vector of length NGROUPS, specifying minimum combined allocations in each group. GroupMax (required). Scalar or a vector of length NGROUPS, specifying maximum combined allocations in each group. See pcglims.

Constraint Type	Description	Values
GroupComparison	Group-to-group comparison constraints.	GroupA (required). NGROUPS-by-NASSETS matrix specifying first group in the comparison. AtoBmin (required). Scalar or vector of length NGROUPS specifying minimum ratios of allocations in GroupA to allocations in GroupB. AtoBmax (required). Scalar or vector of length NGROUPS specifying maximum ratios of allocations in GroupA to allocations in GroupB. GroupB (required). NGROUPS-by-NASSETS matrix specifying second group in the comparison. See pcgcomp .
Custom	Custom linear inequality constraints A*PortWts' <= b.	A (required). NCONSTRAINTS-by-NASSETS matrix, specifying weights for each asset in each inequality equation. b (required). Vector of length NCONSTRAINTS specifying the right hand sides of the inequalities.

Examples

Constrain a portfolio of three assets:

Asset	IBM	HPQ	XOM
Group	A	A	B
Min. Wt.	0	0	0
Max. Wt.	0.5	0.9	0.8

```
NumAssets = 3;
PVal = 1; % Scale portfolio value to 1.
AssetMin = 0;
AssetMax = [0.5 0.9 0.8];
GroupA = [1 1 0];
GroupB = [0 0 1];
AtoBmax = 1.5 % Value of assets in Group A at most 1.5 times value
              % in group B.

ConSet = portcons('PortValue', PVal, NumAssets,'AssetLims',...
AssetMin, AssetMax, NumAssets, 'GroupComparison',GroupA, NaN,...
AtoBmax, GroupB)

ConSet =

    1.0000    1.0000    1.0000    1.0000
   -1.0000   -1.0000   -1.0000   -1.0000
    1.0000         0         0    0.5000
         0    1.0000         0    0.9000
         0         0    1.0000    0.8000
   -1.0000         0         0         0
         0   -1.0000         0         0
         0         0   -1.0000         0
    1.0000    1.0000   -1.5000         0
```

Portfolio weights of 30% in IBM, 30% in HPQ, and 40% in XOM satisfy the constraints.

See Also

`pcalims`, `pcgcomp`, `pcglims`, `pcpval`, `portopt`

Purpose Portfolios on constrained efficient frontier

Syntax

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

Arguments

`ExpReturn`	1 by number of assets (`NASSETS`) vector specifying the expected (mean) return of each asset.
`ExpCovariance`	`NASSETS`-by-`NASSETS` matrix specifying the covariance of the asset returns.
`NumPorts`	(Optional) Number of portfolios generated along the efficient frontier. Returns are equally spaced between the maximum possible return and the minimum risk point. If `NumPorts` is empty (entered as `[]`), computes 10 equally spaced points.
`PortReturn`	(Optional) Expected return of each portfolio. A number of portfolios (`NPORTS`) by 1 vector. If not entered or empty, `NumPorts` equally spaced returns between the minimum and maximum possible values are used.
`ConSet`	(Optional) Constraint matrix for a portfolio of asset investments, created using `portcons`. If not specified, a default is created.

Description

`[PortRisk, PortReturn, PortWts] = portopt(ExpReturn, ExpCovariance, NumPorts, PortReturn, ConSet)` returns the mean-variance efficient frontier with user-specified covariance, returns, and asset constraints (`ConSet`). Given a collection of `NASSETS` risky assets, computes a portfolio of asset investment weights that minimize the risk for given values of the expected return. The portfolio risk is minimized subject to constraints on the total portfolio value, the individual asset minimum and maximum allocation, the asset group minimum and maximum allocation, or the asset group-to-group comparison.

`PortRisk` is an `NPORTS`-by-1 vector of the standard deviation of each portfolio.

`PortReturn` is an `NPORTS`-by-1 vector of the expected return of each portfolio.

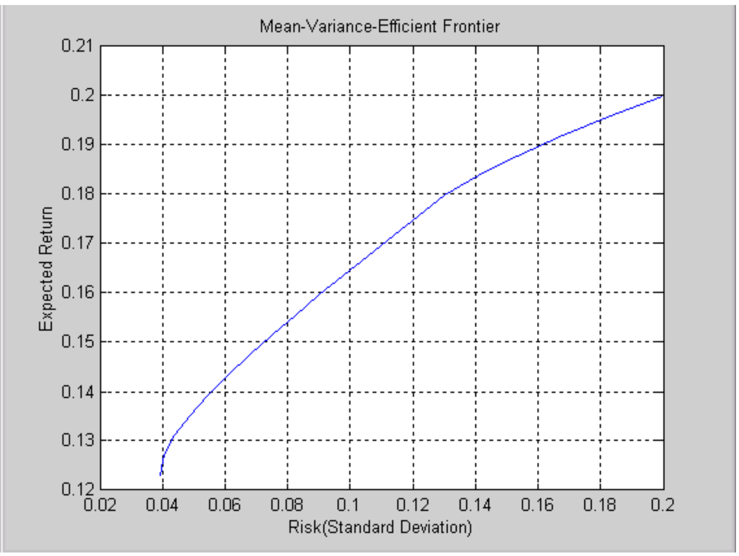
`PortWts` is an `NPORTS`-by-`NASSETS` matrix of weights allocated to each asset. Each row represents a portfolio. The total of all weights in a portfolio is 1.

If portopt is invoked without output arguments, it returns a plot of the efficient frontier.

Examples

Plot the risk-return efficient frontier of portfolios allocated among three assets. Connect 20 portfolios along the frontier having evenly spaced returns. By default, choose among portfolios without short-selling and scale the value of the portfolio to 1.

```
ExpReturn = [0.1 0.2 0.15];

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

NumPorts = 20;
portopt(ExpReturn, ExpCovariance, NumPorts)
```



Return the two efficient portfolios that have returns of 16% and 17%. Limit to portfolios that have at least 20% of the allocation in the first asset, and cap the total value in the first and third assets at 50% of the portfolio.

```
ExpReturn = [0.1 0.2 0.15];

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

PortReturn = [0.16
              0.17];

NumAssets = 3;

AssetMin = [0.20 NaN NaN];

Group    = [1    0   1];

GroupMax = 0.50;

ConSet = portcons('Default', NumAssets, 'AssetLims', AssetMin,...
NaN,'GroupLims', Group, NaN, GroupMax);

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

PortRisk =

    0.0919
    0.1138

PortReturn =

    0.1600
    0.1700

PortWts =

    0.3000    0.5000    0.2000
    0.2000    0.6000    0.2000
```

See Also

ewstats, frontcon, portcons, portstats

Purpose Randomized portfolio risks, returns, and weights

Syntax

```
[PortRisk, PortReturn, PortWts] = portrand(Asset, Return, Points)
portrand(Asset, Return, Points)
```

Arguments

`Asset`	Matrix of time series data. Each row is an observation and each column represents a single security.
`Return`	(Optional) Row vector where each column represents the rate of return for the corresponding security in `Asset`. By default, `Return` is computed by taking the average value of each column of `Asset`.
`Points`	(Optional) Scalar that specifies how many random points should be generated. Default = 1000.

Description

`[PortRisk, PortReturn, PortWts] = portrand(Asset, Return, Points)` returns the risks, rates of return, and weights of random portfolio configurations.

`PortRisk`	`Points`-by-1 vector of standard deviations.
`PortReturn`	`Points`-by-1 vector of expected rates of return.
`PortWts`	`Points` by number of securities matrix of asset weights. Each row of `PortWts` is a different portfolio configuration.

`portrand(Asset, Return, Points)` plots the points representing each portfolio configuration. It does not return any data to the MATLAB workspace.

See Also `frontcon`

References Bodie, Kane, and Marcus, *Investments*, Chapter 7.

portsim

Purpose Monte Carlo simulation of correlated asset returns

Syntax

```
RetSeries = portsim(ExpReturn, ExpCovariance, NumObs, RetIntervals,
   NumSim, Method)
```

Arguments

ExpReturn	1 by number of assets (NASSETS) vector specifying the expected (mean) return of each asset.
ExpCovariance	NASSETS-by-NASSETS matrix of asset return covariances. ExpCovariance must be symmetric and positive semidefinite (no negative eigenvalues). The standard deviations of the returns are: ExpSigma = sqrt(diag(ExpCovariance)).
NumObs	Positive scalar integer indicating the number of consecutive observations in the return time series. If NumObs is entered as the empty matrix [], the length of RetIntervals is used.
RetIntervals	(Optional) Positive scalar or number of observations (NUMOBS) by 1 vector of interval times between observations. If RetIntervals is not specified, all intervals are assumed to have length 1.
NumSim	(Optional) Positive scalar integer indicating the number of simulated sample paths (realizations) of NUMOBS observations. Default = 1 (single realization of NUMOBS correlated asset returns).

Method	(Optional) String indicating the type of Monte Carlo simulation: 'Exact' (default) generates correlated asset returns in which the sample mean and covariance match the input mean (ExpReturn) and covariance (ExpCovariance) specifications. 'Expected' generates correlated asset returns in which the sample mean and covariance are statistically equal to the input mean and covariance specifications. (The expected value of the sample mean and covariance are equal to the input mean (ExpReturn) and covariance (ExpCovariance) specifications.) For either method the sample mean and covariance returned are appropriately scaled by RetIntervals.

Description

portsim simulates correlated returns of NASSETS assets over NUMOBS consecutive observation intervals. Asset returns are simulated as the proportional increments of constant drift, constant volatility stochastic processes, thereby approximating continuous-time geometric Brownian motion.

RetSeries is a NUMOBS-by-NASSETS-by-NUMSIM three-dimensional array of correlated, normally distributed, proportional asset returns. Asset returns over an interval of length dt are given by

$$\frac{dS}{S} = \mu dt + \sigma dz = \mu dt + \sigma\varepsilon\sqrt{dt}$$

where S is the asset price, μ is the expected rate of return, σ is the volatility of the asset price, and ε represents a random drawing from a standardized normal distribution.

Notes 1. When *Method* is 'Exact', the sample mean and covariance of all realizations (scaled by RetIntervals) match the input mean and covariance. When the returns are subsequently converted to asset prices, all terminal prices for a given asset are in close agreement. Although all realizations are drawn independently, they produce similar terminal asset prices. Set *Method* to 'Expected' to avoid this behavior.

2. The returns from the portfolios in PortWts are given by PortReturn = PortWts * RetSeries(:,:,1)', where PortWts is a matrix in which each row contains the asset allocations of a portfolio. Each row of PortReturn corresponds to one of the portfolios identified in PortWts, and each column corresponds to one of the observations taken from the first realization (the first plane) in RetSeries. See portopt and portstats for portfolio specification and optimization.

Examples

Example 1. Distinction Between Simulation Methods

This example highlights the distinction between the Exact and Expected methods of simulation.

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

```
ExpReturn     = [0.0246  0.0189  0.0273  0.0141  0.0311]/100;
Sigmas        = [0.9509  1.4259  1.5227  1.1062  1.0877]/100;
Correlations  = [1.0000  0.4403  0.4735  0.4334  0.6855
                 0.4403  1.0000  0.7597  0.7809  0.4343
                 0.4735  0.7597  1.0000  0.6978  0.4926
                 0.4334  0.7809  0.6978  1.0000  0.4289
                 0.6855  0.4343  0.4926  0.4289  1.0000];
```

Convert the correlations and standard deviations to a covariance matrix.

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

ExpCovariance =

  1.0e-003 *

    0.0904    0.0597    0.0686    0.0456    0.0709
    0.0597    0.2033    0.1649    0.1232    0.0674
    0.0686    0.1649    0.2319    0.1175    0.0816
    0.0456    0.1232    0.1175    0.1224    0.0516
    0.0709    0.0674    0.0816    0.0516    0.1183
```

Assume that there are 252 trading days in a calendar year, and simulate two sample paths (realizations) of daily returns over a two-year period. Since `ExpReturn` and `ExpCovariance` are expressed on a daily basis, set `RetIntervals = 1`.

```
StartPrice    = 100;
NumObs        = 504;   % two calendar years of daily returns
NumSim        = 2;
RetIntervals  = 1;     % one trading day
NumAssets     = 5;
```

To illustrate the distinction between methods, simulate two paths by each method, starting with the same random number state.

```
randn('state',0);
RetExact = portsim(ExpReturn, ExpCovariance, NumObs, ...
RetIntervals, NumSim, 'Exact');
randn('state',0);
RetExpected = portsim(ExpReturn, ExpCovariance, NumObs, ...
RetIntervals, NumSim, 'Expected');
```

If you compare the mean and covariance of `RetExact` with the inputs (`ExpReturn` and `ExpCovariance`), you will observe that they are almost identical.

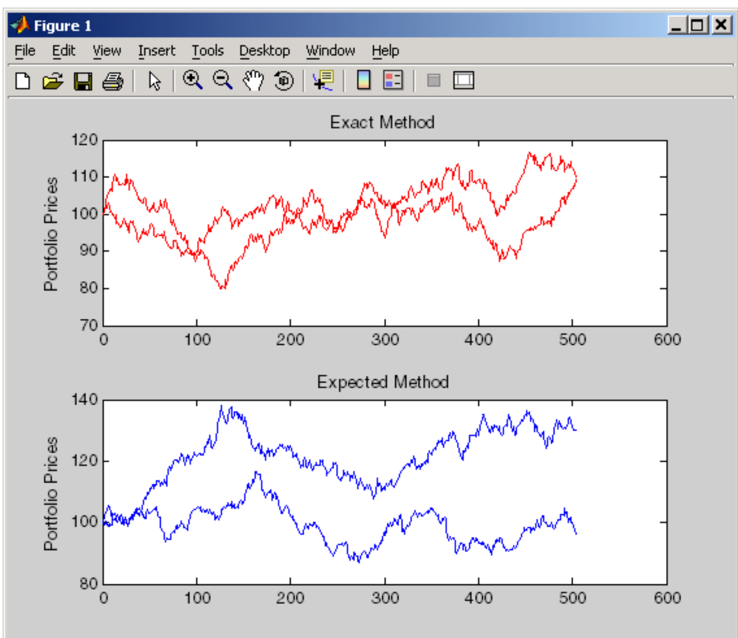
At this point, `RetExact` and `RetExpected` are both 504-by-5-by-2 arrays. Now assume an equally-weighted portfolio formed from the five assets and create arrays of portfolio returns in which each column represents the portfolio return of the corresponding sample path of the simulated returns of the five assets. The portfolio arrays `PortRetExact` and `PortRetExpected` are 504-by-2 matrices.

```
Weights         = ones(NumAssets, 1)/NumAssets;
PortRetExact    = zeros(NumObs, NumSim);
PortRetExpected = zeros(NumObs, NumSim);

for i = 1:NumSim
    PortRetExact(:,i)    = RetExact(:,:,i) * Weights;
    PortRetExpected(:,i) = RetExpected(:,:,i) * Weights;
end
```

Finally, convert the simulated portfolio returns to prices and plot the data. In particular, note that since the Exact method matches expected return and covariance, the terminal portfolio prices are virtually identical for each sample path. This is not true for the Expected simulation method.

Although this example examines portfolios, the same methods apply to individual assets as well. Thus, Exact simulation is most appropriate when unique paths are required to reach the same terminal prices.

```
PortExact   = ret2tick(PortRetExact, ...
repmat(StartPrice,1,NumSim));
PortExpected = ret2tick(PortRetExpected, ...
repmat(StartPrice,1,NumSim));
subplot(2,1,1), plot(PortExact, '-r')
ylabel('Portfolio Prices')
title('Exact Method')
subplot(2,1,2), plot(PortExpected, '-b')
ylabel('Portfolio Prices')
title('Expected Method')
```



Example 2. Interaction between `ExpReturn`, `ExpCovariance` and `RetIntervals`

Recall that `portsim` simulates correlated asset returns over an interval of length dt, given by the equation

$$\frac{dS}{S} = \mu dt + \sigma dz = \mu dt + \sigma\varepsilon\sqrt{dt}$$

where S is the asset price, μ is the expected rate of return, σ is the volatility of the asset price, and ε represents a random drawing from a standardized normal distribution.

The time increment dt is determined by the optional input `RetIntervals`, either as an explicit input argument or as a unit time increment by default. Regardless, the periodicity of `ExpReturn`, `ExpCovariance` and `RetIntervals` must be consistent. For example, if `ExpReturn` and `ExpCovariance` are annualized, then `RetIntervals` must be in years. This point is often misunderstood.

To illustrate the interplay among `ExpReturn`, `ExpCovariance`, and `RetIntervals`, consider a portfolio of five assets with the following expected returns, standard deviations, and correlation matrix based on daily asset returns.

```
ExpReturn     = [0.0246  0.0189  0.0273  0.0141  0.0311]/100;

Sigmas        = [0.9509  1.4259  1.5227  1.1062  1.0877]/100;

Correlations  = [1.0000  0.4403  0.4735  0.4334  0.6855
                 0.4403  1.0000  0.7597  0.7809  0.4343
                 0.4735  0.7597  1.0000  0.6978  0.4926
                 0.4334  0.7809  0.6978  1.0000  0.4289
                 0.6855  0.4343  0.4926  0.4289  1.0000];
```

Convert the correlations and standard deviations to a covariance matrix of daily returns.

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

Assume 252 trading days per calendar year, and simulate a single sample path of daily returns over a four-year period. Since the `ExpReturn` and `ExpCovariance` inputs are expressed on a daily basis, set `RetIntervals = 1`.

```
StartPrice    = 100;
NumObs        = 1008;    % four calendar years of daily returns
RetIntervals  = 1;       % one trading day
NumAssets     = length(ExpReturn);
randn('state',0);
RetSeries1 = portsim(ExpReturn, ExpCovariance, NumObs, ...
RetIntervals, 1, 'Expected');
```

Now annualize the daily data, thereby changing the periodicity of the data, by multiplying `ExpReturn` and `ExpCovariance` by 252 and dividing `RetIntervals` by 252 (`RetIntervals` = 1/252 of a year).

Resetting the random number generator to its initial state, you can reproduce the results.

```
randn('state',0);
RetSeries2 = portsim(ExpReturn*252, ExpCovariance*252, ...
NumObs, RetIntervals/252, 1, 'Expected');
```

Assume an equally-weighted portfolio and compute portfolio returns associated with each simulated return series.

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

PortRet1 = RetSeries2 * Weights;
PortRet2 = RetSeries2 * Weights;
```

Comparison of the data reveals that `PortRet1` and `PortRet2` are identical.

Example 3. Univariate Geometric Brownian Motion

This example simulates a univariate geometric Brownian motion process. It is based on an example found in Hull, *Options, Futures, and Other Derivatives*, 5th Edition. (See example 12.2 on page 236). In addition to verifying Hull's example, it also graphically illustrates the lognormal property of terminal stock prices by a rather large Monte Carlo simulation.

First, assume you own a stock with an initial price of $20, an annualized expected return of 20% and volatility of 40%. Simulate the daily price process for this stock over the course of one full calendar year (252 trading days).

```
StartPrice    = 20;
ExpReturn     = 0.2;
```

```
ExpCovariance = 0.4^2;
NumObs        = 252;
NumSim        = 10000;
RetIntervals  = 1/252;
```

Note that `RetIntervals` is expressed in years, consistent with the fact that `ExpReturn` and `ExpCovariance` are annualized. Also, note that `ExpCovariance` is entered as a variance rather than the more familiar standard deviation (volatility).

Now set the random number generator state, and simulate 10,000 trials (realizations) of stock returns over a full calendar year of 252 trading days.

```
randn('state',10);
RetSeries = squeeze(portsim(ExpReturn, ExpCovariance, NumObs, ...
RetIntervals, NumSim, 'Expected'));
```

The `squeeze` function simply reformats the output array of simulated returns from a 252-by-1-by-10000 array to more convenient 252-by-10000 array. (Recall that `portsim` is fundamentally a multivariate simulation engine).

In accordance with Hull's equations 12.4 and 12.5 on page 236

$$E(S_T) = S_0 e^{\mu T}$$

$$var(S_T) = S_0^2 e^{2\mu T}(e^{\sigma^2 T} - 1)$$

convert the simulated return series to a price series and compute the sample mean and the variance of the terminal stock prices.

```
StockPrices = ret2tick(RetSeries, repmat(StartPrice, 1, NumSim));

SampMean = mean(StockPrices(end,:))

SampMean =

   24.4587
```

```
SampVar = var(StockPrices(end,:))

SampVar =

  104.2016
```

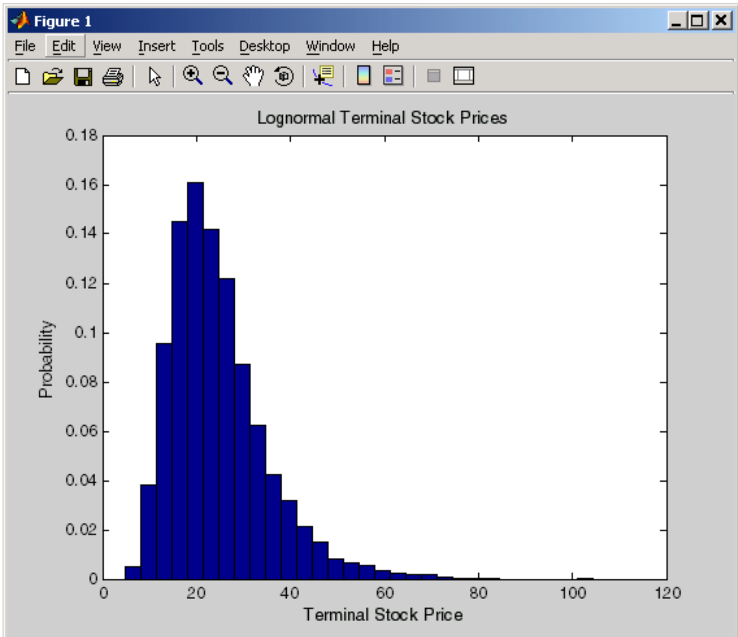
Compare these values with the values you obtain by using Hull's equations.

```
ExpValue = StartPrice*exp(ExpReturn)

ExpValue =

   24.4281

ExpVar = ...
StartPrice*StartPrice*exp(2*ExpReturn)*(exp((ExpCovariance)) - 1)

ExpVar =

  103.5391
```

These results are very close to the results shown in Hull's example 12.2.

Next, display the sample density function of the terminal stock price after one calendar year. From the sample density function, the lognormal distribution of terminal stock prices is apparent.

```
[count, BinCenter] = hist(StockPrices(end,:), 30);
figure
bar(BinCenter, count/sum(count), 1, 'r')
xlabel('Terminal Stock Price')
ylabel('Probability')
title('Lognormal Terminal Stock Prices')
```



See Also

`ewstats`, `portopt`, `portstats`, `randn`, `ret2tick`

References

Hull, John, C., *Options, Futures, and Other Derivatives*, Upper Saddle River, New Jersey: Prentice-Hall. 5th ed., 2003, ISBN 0-13-009056-5.

portstats

Purpose Portfolio expected return and risk

Syntax

```
[PortRisk, PortReturn] = portstats(ExpReturn, ExpCovariance,
  PortWts)
```

Arguments

ExpReturn	1 by number of assets (NASSETS) vector specifying the expected (mean) return of each asset.
ExpCovariance	NASSETS-by-NASSETS matrix specifying the covariance of the asset returns.
PortWts	(Optional) Number of portfolios (NPORTS) by NASSETS matrix of weights allocated to each asset. Each row represents a different weighting combination. Default = 1/NASSETS (equally weighted).

Description

[PortRisk, PortReturn] = portstats(ExpReturn, ExpCovariance, PortWts) computes the expected rate of return and risk for a portfolio of assets.

PortRisk is an NPORTS-by-1 vector of the standard deviation of each portfolio.

PortReturn is an NPORTS-by-1 vector of the expected return of each portfolio.

Examples

```
ExpReturn = [0.1 0.2 0.15];

ExpCovariance = [0.0100   -0.0061    0.0042
                -0.0061    0.0400   -0.0252
                 0.0042   -0.0252    0.0225 ];

PortWts=[0.4 0.2 0.4; 0.2 0.4 0.2];

[PortRisk, PortReturn] = portstats(ExpReturn, ExpCovariance,...
PortWts)

PortRisk =

    0.0560
    0.0550
```

```
PortReturn =

    0.1400
    0.1300
```

See Also

`frontcon`

portvrisk

Purpose

Portfolio value at risk

Syntax

```
ValueAtRisk = portvrisk(PortReturn, PortRisk, RiskThreshold,
   PortValue)
```

Arguments

`PortReturn`	Number of portfolios (`NPORTS`) by 1 vector or scalar of the expected return of each portfolio over the period.
`PortRisk`	`NPORTS`-by-1 vector or scalar of the standard deviation of each portfolio over the period.
`RiskThreshold`	(Optional) `NPORTS`-by-1 vector or scalar specifying the loss probability. Default = 0.05 (5%).
`PortValue`	(Optional) `NPORTS`-by-1 vector or scalar specifying the total value of asset portfolio. Default = 1.

Description

`ValueAtRisk = portvrisk(PortReturn, PortRisk, RiskThreshold, PortValue)` returns the maximum potential loss in the value of a portfolio over one period of time, given the loss probability level `RiskThreshold`.

`ValueAtRisk` is an `NPORTS`-by-1 vector of the estimated maximum loss in the portfolio, predicted with a confidence probability of 1- `RiskThreshold`.

If `PortValue` is not given, `ValueAtRisk` is presented on a per-unit basis. A value of 0 indicates no losses.

Examples

This example computes `ValueAtRisk` on a per-unit basis.

```
PortReturn = 0.29/100;
PortRisk = 3.08/100;
RiskThreshold = [0.01;0.05;0.10];
PortValue = 1;
ValueAtRisk = portvrisk(PortReturn,PortRisk,...
RiskThreshold,PortValue)
ValueAtRisk =

    0.0688
    0.0478
    0.0366
```

This example computes `ValueAtRisk` with actual values.

```
PortReturn = [0.29/100;0.30/100];
PortRisk = [3.08/100;3.15/100];
RiskThreshold = 0.10;
PortValue = [1000000000;500000000];
ValueAtRisk = portvrisk(PortReturn,PortRisk,...
RiskThreshold,PortValue)
ValueAtRisk =

  1.0e+007 *
    3.6572
    1.8684
```

See Also

`frontcon`, `portopt`

Purpose Price bonds in a portfolio by a set of zero curves

Syntax

```
BondPrices = prbyzero(Bonds, Settle, ZeroRates, ZeroDates)
```

Arguments

`Bonds` Coupon bond information used to compute prices. A number of bonds (`NUMBONDS`) by 6 matrix where each row describes a bond. The first two columns are required; the rest are optional but must be added in order. All rows in `Bonds` must have the same number of columns. Columns are `[Maturity CouponRate Face Period Basis EndMonthRule]` where:

`Maturity`	Maturity date as a serial date number or date string
`CouponRate`	Decimal number indicating the annual percentage rate used to determine the coupons payable on a bond
`Face`	(Optional) Face or par value of the bond. Default = 100.
`Period`	(Optional) Coupons per year of the bond. Allowed values are 0,1, 2 (default), 3, 4, 6, and 12.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. 0 = actual/actual (default), 1 = 30/360 (SIA), 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).
`EndMonthRule`	(Optional) End-of-month rule. This rule applies only when `Maturity` is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond's coupon payment date is always the same numerical day of the month. 1 = set rule on (default), meaning that a bond's coupon payment date is always the last actual day of the month.

`Settle`	Serial date number of the settlement date.
`ZeroRates`	`NUMDATES`-by-`NUMCURVES` matrix of observed zero rates, as decimal fractions. Each column represents a rate curve. Each row represents an observation date.
`ZeroDates`	`NUMDATES`-by-1 column of dates for observed zeros

Description

`BondPrices = prbyzero(Bonds, Settle, ZeroRates, ZeroDates)` computes the bond prices in a portfolio using a set of zero curves.

`BondPrices` is a `NUMBONDS`-by-`NUMCURVES` matrix of clean bond prices. Each column is derived from the corresponding zero curve in `ZeroRates`.

Examples

This example uses `zbtprice` to compute a zero curve given a portfolio of coupon bonds and their prices. It then reverses the process, using the zero curve as input to `prbyzero` to compute the prices.

```
Bonds = [datenum('6/1/1998') 0.0475 100 2 0 0;
         datenum('7/1/2000') 0.06 100 2 0 0;
         datenum('7/1/2000') 0.09375 100 6 1 0;
         datenum('6/30/2001') 0.05125 100 1 3 1;
         datenum('4/15/2002') 0.07125 100 4 1 0;
         datenum('1/15/2000') 0.065 100 2 0 0;
         datenum('9/1/1999') 0.08 100 3 3 0;
         datenum('4/30/2001') 0.05875 100 2 0 0;
         datenum('11/15/1999') 0.07125 100 2 0 0;
         datenum('6/30/2000') 0.07 100 2 3 1;
         datenum('7/1/2001') 0.0525 100 2 3 0;
         datenum('4/30/2002') 0.07 100 2 0 0];

Prices = [ 99.375;
           99.875;
          105.75 ;
           96.875;
          103.625;
          101.125;
          103.125;
           99.375;
          101.0  ;
          101.25 ;
```

```
           96.375;
          102.75 ];

Settle = datenum('12/18/1997');
```

Set semiannual compounding for the zero curve, on an actual/365 basis. Derive the zero curve within 50 iterations.

```
OutputCompounding = 2;
OutputBasis = 3;
MaxIterations = 50;
```

Execute zbtprice

```
[ZeroRates, ZeroDates] = zbtprice(Bonds, Prices, Settle,...
OutputCompounding, OutputBasis, MaxIterations)
```

which returns the zero curve at the maturity dates.

```
ZeroRates =

            0.0616
            0.0609
            0.0658
            0.0590
            0.0648
            0.0655
            0.0606
            0.0601
            0.0642
            0.0621
            0.0627

ZeroDates =

             729907
             730364
             730439
             730500
             730667
             730668
             730971
```

```
731032
731033
731321
731336
```

Now execute `prbyzero`

```
BondPrices = prbyzero(Bonds, Settle, ZeroRates, ZeroDates)
```

which returns

```
BondPrices =

   99.38
   98.80
  106.83
   96.88
  103.62
  101.13
  103.12
   99.36
  101.00
  101.25
   96.37
  102.74
```

In this example `zbtprice` and `prbyzero` do not exactly reverse each other. Many of the bonds have the end-of-month rule off (`EndMonthRule = 0`). The rule subtly affects the time factor computation. If you set the rule on (`EndMonthRule = 1`) everywhere in the `Bonds` matrix, then `prbyzero` returns the original prices, except when the two incompatible prices fall on the same maturity date.

See Also

`tr2bonds`, `zbtprice`

prdisc

Purpose

Price of discounted security

Syntax

```
Price = prdisc(Settle, Maturity, Face, Discount, Basis)
```

Arguments

`Settle`	Enter as serial date number or date string. `Settle` must be earlier than or equal to `Maturity`.
`Maturity`	Enter as serial date number or date string.
`Face`	Redemption (par, face) value.
`Discount`	Bank discount rate of the security. Enter as decimal fraction.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. 0 = actual/actual (default), 1 = 30/360 (SIA), 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).

Description

`Price = prdisc(Settle, Maturity, Face, Discount, Basis)` returns the price of a security whose yield is quoted as a bank discount rate (e.g., U. S. Treasury Bills).

Examples

Using this data

```
Settle = '10/14/2000';
Maturity = '03/17/2001';
Face = 100;
Discount = 0.087;
Basis = 2;

Price = prdisc(Settle, Maturity, Face, Discount, Basis)
```

returns

```
 Price =

         96.2783
```

See Also

`acrudisc`, `bndprice`, `discrate`, `prmat`, `ylddisc`

References

Mayle, *Standard Securities Calculation Methods*, Volumes I-II, 3rd edition. Formula 2.

prmat

Purpose

Price with interest at maturity

Syntax

```
[Price, AccruInterest] = prmat(Settle, Maturity, Issue, Face,
  CouponRate, Yield, Basis)
```

Arguments

Settle	Enter as serial date number or date string. Settle must be earlier than or equal to Maturity.
Maturity	Enter as serial date number or date string.
Issue	Enter as serial date number or date string.
Face	Redemption (par, face) value.
CouponRate	Enter as decimal fraction.
Yield	Annual yield. Enter as decimal fraction.
Basis	(Optional) Day-count basis of the instrument. A vector of integers. 0 = actual/actual (default), 1 = 30/360 (SIA), 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).

Description

[Price, AccruInterest] = prmat(Settle, Maturity, Issue, Face, CouponRate, Yield, Basis) returns the price and accrued interest of a security that pays interest at maturity. This function also applies to zero-coupon bonds or pure discount securities by setting CouponRate = 0.

Examples

Using this data

```
Settle = '02/07/2002';
Maturity = '04/13/2002';
Issue = '10/11/2001';
Face = 100;
CouponRate = 0.0608;
Yield = 0.0608;
Basis = 1;

[Price, AccruInterest] = prmat(Settle, Maturity, Issue, Face,...
CouponRate, Yield, Basis)
```

returns

```
Price =

    99.9784

AccruInterest =

     1.9591
```

See Also

`acrubond`, `acrudisc`, `bndprice`, `prdisc`, `yldmat`

References

Mayle, *Standard Securities Calculation Methods*, Volumes I-II, 3rd edition. Formula 4.

prtbill

Purpose Price of Treasury bill

Syntax

```
Price = prtbill(Settle, Maturity, Face, Discount)
```

Arguments

`Settle`	Enter as serial date number or date string. `Settle` must be earlier than or equal to `Maturity`.
`Maturity`	Enter as serial date number or date string.
`Face`	Redemption (par, face) value.
`Discount`	Discount rate of the Treasury bill. Enter as decimal fraction.

Description `Price = prtbill(Settle, Maturity, Face, Discount)` returns the price for a Treasury bill.

Examples The settlement date of a Treasury bill is February 10, 2002, the maturity date is August 6, 2002, the discount rate is 3.77%, and the par value is $1000. Using this data

```
Price = prtbill('2/10/2002', '8/6/2002', 1000, 0.0377)
```

returns

```
Price =
        981.4642
```

See Also `beytbill`, `yldtbill`

References Bodie, Kane, and Marcus, *Investments*, pages 41-43.

Purpose Present value with fixed periodic payments

Syntax

```
PresentVal = pvfix(Rate, NumPeriods, Payment, ExtraPayment, Due)
```

Arguments

`rate`	Periodic interest rate, as a decimal fraction.
`NumPeriods`	Number of periods.
`Payment`	Periodic payment.
`ExtraPayment`	(Optional) Payment received other than `Payment` in the last period. Default = 0.
`Due`	(Optional) When payments are due or made: 0 = end of period (default), or 1 = beginning of period.

Description `PresentVal = pvfix(Rate, NumPeriods, Payment, ExtraPayment, Due)` returns the present value of a series of equal payments.

Examples $200 is paid monthly into a savings account earning 6%. The payments are made at the end of the month for five years. To find the present value of these payments

```
PresentVal = pvfix(0.06/12, 5*12, 200, 0, 0)
```

returns

```
PresentVal =

          10345.11
```

See Also `fvfix`, `fvvar`, `payper`, `pvvar`

Purpose

Present value of varying cash flow

Syntax

```
PresentVal = pvvar(CashFlow, Rate, IrrCFDates)
```

Arguments

`CashFlow`	A vector of varying cash flows. Include the initial investment as the initial cash flow value (a negative number).
`Rate`	Periodic interest rate. Enter as a decimal fraction.
`IrrCFDates`	(Optional) For irregular (nonperiodic) cash flows, a vector of dates on which the cash flows occur. Enter dates as serial date numbers or date strings. Default assumes `CashFlow` contains regular (periodic) cash flows.

Description

`PresentVal = pvvar(CashFlow, Rate, IrrCFDates)` returns the net present value of a varying cash flow.

Examples

This cash flow represents the yearly income from an initial investment of $10,000. The annual interest rate is 8%.

Year 1	$2000
Year 2	$1500
Year 3	$3000
Year 4	$3800
Year 5	$5000

To calculate the net present value of this regular cash flow

```
PresentVal = pvvar([-10000 2000 1500 3000 3800 5000], 0.08)
```

returns

```
PresentVal =

          1715.39
```

An investment of $10,000 returns this irregular cash flow. The original investment and its date are included. The periodic interest rate is 9%.

Cash flow	Dates
($10000)	January 12, 1987
$2500	February 14, 1988
$2000	March 3, 1988
$3000	June 14, 1988
$4000	December 1, 1988

To calculate the net present value of this irregular cash flow

```
CashFlow = [-10000, 2500, 2000, 3000, 4000];

IrrCFDates = ['01/12/1987'
              '02/14/1988'
              '03/03/1988'
              '06/14/1988'
              '12/01/1988'];

PresentVal = pvvar(CashFlow, 0.09, IrrCFDates)
```

returns

```
PresentVal =

             142.16
```

See Also

fvfix, fvvar, irr, payuni, pvfix

pyld2zero

Purpose Zero curve given a par yield curve

Syntax

```
[ZeroRates, CurveDates] = pyld2zero(ParRates, CurveDates, Settle,
   Compounding, Basis, OutputCompounding)
```

Arguments

`ParRates`	Column vector of annualized implied par yield rates, as decimal fractions. (Par yields = coupon rates.) In aggregate, the yield rates in `ParRates` constitute an implied par yield curve for the investment horizon represented by `CurveDates`.
`CurveDates`	Column vector of maturity dates (as serial date numbers) that correspond to the par rates.
`Settle`	A serial date number that is the common settlement date for the par rates.
`Compounding`	(Optional) A scalar that sets the rate at which the par rates are compounded when annualized. Allowed values are: 1 annual compounding 2 semiannual compounding (default) 3 compounding three times per year 4 quarterly compounding 6 bimonthly compounding 12 monthly compounding 365 daily compounding -1 continuous compounding
`Basis`	(Optional) Day-count basis used to annualize the zero rates. 0 = actual/actual (default), 1 = 30/360 (SIA), 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).
`OutputCompounding`	(Optional) Value representing the rate at which the zero rates are compounded. Default = `Compounding`.

Description

`[ZeroRates, CurveDates] = pyld2zero(ParRates, CurveDates, Settle, Compounding, Basis, OutputCompounding)` returns a zero curve given a par yield curve and its maturity dates.

`ZeroRates`	Column vector of decimal fractions. In aggregate, the rates in `ZeroRates` constitute a zero curve for the investment horizon represented by `CurveDates`.
`CurveDates`	Column vector of maturity dates (as serial date numbers) corresponding to the zero rates. This vector is the same as the input vector `CurveDates`.

Examples

Given

- A par yield curve over a set of maturity dates
- A settlement date
- Annual compounding for the input par rates and monthly compounding for the output zero curve

compute a zero yield curve.

```
ParRates = [0.0479
            0.0522
            0.0540
            0.0540
            0.0536
            0.0532
            0.0532
            0.0539
            0.0558
            0.0543];

CurveDates = [datenum('06-Nov-2000')
      datenum('11-Dec-2000')
      datenum('15-Jan-2001')
      datenum('05-Feb-2001')
      datenum('04-Mar-2001')
      datenum('02-Apr-2001')
      datenum('30-Apr-2001')
      datenum('25-Jun-2001')
```

```
      datenum('04-Sep-2001')
      datenum('12-Nov-2001')];

Settle = datenum('03-Nov-2000');
Compounding = 1;
OutputCompounding = 12;

[ZeroRates, CurveDates] = pyld2zero(ParRates, CurveDates,...
Settle, Compounding, [], OutputCompounding)

ZeroRates =

    0.0484
    0.0529
    0.0549
    0.0550
    0.0547
    0.0544
    0.0545
    0.0551
    0.0572
    0.0557

CurveDates =

      730796
      730831
      730866
      730887
      730914
      730943
      730971
      731027
      731098
      731167
```

For readability, `ParRates` and `ZeroRates` are shown only to the basis point. However, MATLAB computes them at full precision. If you enter `ParRates` as shown, `ZeroRates` may differ due to rounding.

See Also `zero2pyld` and other functions for Term Structure of Interest Rates

ret2tick

Purpose

Convert a return series to a price series

Syntax

```
[TickSeries, TickTimes] = ret2tick(RetSeries, StartPrice,
   RetIntervals, StartTime, Method)
```

Arguments

`RetSeries`	Number of observations (`NUMOBS`) by number of assets (`NASSETS`) time series array of asset returns associated with the prices in `TickSeries`. The *i*'th return is quoted for the period `TickTimes(i)` to `TickTimes(i+1)` and is not normalized by the time increment between successive price observations.
`StartPrice`	(Optional) 1-by-`NASSETS` vector of initial asset prices or a single scalar initial price applied to all assets. Prices start at 1 if `StartPrice` is not specified.
`RetIntervals`	(Optional) Scalar or `NUMOBS`-by-1 vector of interval times between observations. If this argument is not specified, all intervals are assumed to have length 1.
`StartTime`	(Optional) Starting time for first observation, applied to the price series of all assets. The default is zero.
`Method`	(Optional) Character string indicating the method to convert asset returns to prices. Must be `'Simple'` (default) or `'Continuous'`. If `Method` is `'Simple'`, `ret2tick` uses simple periodic returns. If `Method` is `'Continuous'`, the function uses continuously compounded returns. Case is ignored for *`Method`*.

Description

`[TickSeries, TickTimes] = ret2tick(RetSeries, StartPrice, RetIntervals, StartTime,` *`Method`*`)` generates price values from the starting prices of `NASSETS` investments and `NUMOBS` incremental return observations.

`TickSeries` is a `NUMOBS+1`-by-`NASSETS` times series array of equity prices. The first row contains the oldest observations and the last row the most recent. Observations across a given row occur at the same time for all columns. Each column is a price series of an individual asset. If *`Method`* is unspecified or `'Simple'`, the prices are

```
TickSeries(i+1) = TickSeries(i)*[1 + RetSeries(i)]
```

If *Method* is 'Continuous', the prices are

```
TickSeries(i+1) = TickSeries(i)*exp[RetSeries(i)]
```

TickTimes is a NUMOBS+1 column vector of monotonically increasing observation times associated with the prices in TickSeries. The initial time is zero unless specified in StartTime, and sequential observation times occur at unit increments unless specified in RetIntervals.

Examples

Compute the price increase of two stocks over a year's time based on three incremental return observations.

```
RetSeries = [0.10 0.12
             0.05 0.04
            -0.05 0.05];

RetIntervals = [182
                 91
                 92];

StartTime = datenum('18-Dec-2000');

[TickSeries,TickTimes] = ret2tick(RetSeries,[],RetIntervals,...
StartTime)

TickSeries =

    1.0000    1.0000
    1.1000    1.1200
    1.1550    1.1648
    1.0973    1.2230

TickTimes =

      730838
      731020
      731111
      731203

datestr(TickTimes)
```

```
ans =

18-Dec-2000
18-Jun-2001
17-Sep-2001
18-Dec-2001
```

See Also `portsim`, `tick2ret`

Purpose Seconds of date or time

Syntax

```
Seconds = second(Date)
```

Description `Seconds = second(Date)` returns the seconds given a serial date number or a date string.

Examples

```
Seconds = second(738647.558427893)
```

or

```
Seconds = second('06-May-2022, 13:24:08.17')
```

returns

```
Seconds =

        8.1700
```

See Also `datevec`, `hour`, `minute`

taxedrr

Purpose After-tax rate of return

Syntax

```
Return = taxedrr(PreTaxReturn, TaxRate)
```

Arguments

`PreTaxReturn`	Nominal rate of return. Enter as a decimal fraction.
`TaxRate`	Tax rate. Enter as a decimal fraction.

Description `Return = taxedrr(PreTaxReturn, TaxRate)` calculates the after-tax rate of return.

Examples An investment has a 12% nominal rate of return and is taxed at a 30% rate. The after-tax rate of return is

```
Return = taxedrr(0.12, 0.30)

Return =
          0.0840
```

or 8.4%

See Also `effrr`, `irr`, `mirr`, `nomrr`, `xirr`

Purpose

Treasury bond parameters given Treasury bill parameters

Syntax

```
[TBondMatrix, Settle] = tbl2bond(TBillMatrix)
```

Arguments

`TBillMatrix` Treasury bill parameters. An n-by-5 matrix where each row describes a Treasury bill. n is the number of Treasury bills. Columns are `[Maturity DaysMaturity Bid Asked AskYield]` where:

`Maturity`	Maturity date, as a serial date number. Use `datenum` to convert date strings to serial date numbers.
`DaysMaturity`	Days to maturity, as an integer. Days to maturity is quoted on a skip-day basis; the actual number of days from settlement to maturity is `DaysMaturity + 1`.
`Bid`	Bid bank-discount rate: the percentage discount from face value at which the bill could be bought, annualized on a simple-interest basis. A decimal fraction.
`Asked`	Asked bank-discount rate, as a decimal fraction.
`AskYield`	Asked yield: the bond-equivalent yield from holding the bill to maturity, annualized on a simple-interest basis and assuming a 365-day year. A decimal fraction.

Description

`[TBondMatrix, Settle] = tbl2bond(TBillMatrix)` restates U.S. Treasury bill market parameters in U.S. Treasury bond form as zero-coupon bonds. This function makes Treasury bills directly comparable to Treasury bonds and notes.

`TBondMatrix`	Treasury bond parameters. An N-by-5 matrix where each row describes an equivalent Treasury (zero-coupon) bond. Columns are `[CouponRate Maturity Bid Asked AskYield]` where

`CouponRate`	Coupon rate, which is always 0.
`Maturity`	Maturity date, as a serial date number. This date is the same as the Treasury bill `Maturity` date.
`Bid`	Bid price based on $100 face value.
`Asked`	Asked price based on $100 face value.
`AskYield`	Asked yield to maturity: the effective return from holding the bond to maturity, annualized on a compound-interest basis.

Examples

Given published Treasury bill market parameters for December 22, 1997

```
TBill = [datenum('jan 02 1998')  10  0.0526  0.0522  0.0530
         datenum('feb 05 1998')  44  0.0537  0.0533  0.0544
         datenum('mar 05 1998')  72  0.0529  0.0527  0.0540];
```

Execute the function.

```
TBond = tbl2bond(TBill)

TBond =
            0  729760      99.854      99.855     0.053
            0  729790      99.344      99.349     0.0544
            0  729820      98.942      98.946     0.054
```

(Example output has been formatted for readability.)

See Also

`tr2bonds` and other functions for Term Structure of Interest Rates

Purpose

Find third Wednesday of month

Syntax

```
[BeginDates, EndDates] = thirdwednesday(Month, Year)
```

Arguments

`Month`	Month of delivery for Eurodollar futures.
`Year`	Four-digit year of delivery for Eurodollar futures, in sequence corresponding to a month in the `Month` input argument.

Inputs can be scalars or n-by-1 vectors.

Description

`[BeginDates, EndDates] = thirdwednesday(Month, Year)` computes the beginning and end period date for a LIBOR contract (third Wednesdays of delivery months).

`BeginDates` is the beginning of three-month period contract as specified by `Month` and `Year`.

`EndDates` is the end of three-month period contract as specified by `Month` and `Year`.

Note

1. All dates are returned as serial date numbers. Convert to strings using `datestr`.
2. The function returns duplicates if you supply identical months and years.
3. The function supports dates from January 2000 to December 2099.

Examples

Find the third Wednesday dates for swaps commencing in the month of October in the years 2002, 2003, and 2004.

```
Months = [10; 10; 10];
Year = [2002; 2003; 2004];
[BeginDates, EndDates] = thirdwednesday(Months, Year);
```

```
datestr(BeginDates)

ans =

16-Oct-2002
15-Oct-2003
20-Oct-2004

datestr(EndDates)

ans =

16-Jan-2003
15-Jan-2004
20-Jan-2005
```

Purpose Thirty-second quotation to decimal

Syntax

```
OutNumber =  thirtytwo2dec(InNumber, InFraction)
```

Arguments

`InNumber`	Scalar or vector of input numbers without fractional component.
`InFraction`	Scalar or vector of fractional portions of each element in `InNumber`.

Description

`OutNumber = thirtytwo2dec(InNumber, InFraction)` changes the price quotation for a bond or bond future from a fraction with a denominator of 32 to a decimal.

`OutNumber` represents the sum of `InNumber` and `InFraction` expressed as a decimal.

Examples

Two bonds are quoted as 101-25 and 102-31. Convert these prices to decimal.

```
InNumber  = [101; 102];
InFraction = [25; 31]

OutNumber = thirtytwo2dec(InNumber, InFraction)

OutNumber =

    101.7813
    102.9688
```

See Also `dec2thirtytwo`

Purpose

Convert a price series to a return series

Syntax

```
[RetSeries, RetIntervals] = tick2ret(TickSeries, TickTimes, Method)
```

Arguments

`TickSeries`	Number of observations (`NUMOBS`) by number of assets (`NASSETS`) matrix of prices of equity assets. Each column is a price series of an individual asset. First row is oldest observation. Last row is most recent. Observations across a given row occur at the same time for all columns.
`TickTimes`	(Optional) `NUMOBS`-by-1 increasing vector of observation times associated with the prices in `TickSeries`. Times are serial date numbers (day units) or decimal numbers in arbitrary units (e.g., yearly). If `TickTimes` is empty or missing, sequential observation times from 1, 2, ... `NUMOBS` are assumed.
`Method`	(Optional) Character string indicating the method to convert prices to asset returns. Must be `'Simple'` (default) or `'Continuous'`. If `Method` is `'Simple'`, `tick2ret` computes simple periodic returns. If `Method` is `'Continuous'`, returns are continuously compounded. Case is ignored for *`Method`*.

Description

`[RetSeries, RetIntervals] = tick2ret(TickSeries, TickTimes, `*`Method`*`)` computes the asset returns realized between `NUMOBS` observations of prices of `NASSETS` assets.

RetSeries is a (`NUMOBS-1`)-by-`NASSETS` time series array of asset returns associated with the prices in `TickSeries`. The *i*'th return is quoted for the period `TickTimes(i)` to `TickTimes(i+1)` and is not normalized by the time increment between successive price observations. If *`Method`* is unspecified or `'Simple'`, the returns are:

```
RetSeries(i) = TickSeries(i+1)/TickSeries(i) - 1
```

If *`Method`* is `'Continuous'`, the returns are:

```
RetSeries(i) = log[TickSeries(i+1)/TickSeries(i)]
```

`RetIntervals` is a (`NUMOBS-1`)-by-1 column vector of interval times between observations. If `TickTimes` is empty or unspecified, all intervals are assumed to have length 1.

Examples

Compute the periodic returns of two stocks observed in the first, second, third, and fourth quarters.

```
TickSeries = [100 80
              110 90
              115 88
              110 91];

TickTimes = [0
             6
             9
             12];

[RetSeries, RetIntervals] = tick2ret(TickSeries, TickTimes)

RetSeries =

    0.1000    0.1250
    0.0455   -0.0222
   -0.0435    0.0341

RetIntervals =

     6
     3
     3
```

See Also

`ewstats`, `ret2tick`

time2date

Purpose Dates from time and frequency

Syntax

```
Dates = time2date(Settle, TFactors, Compounding, Basis,
   EndMonthRule)
```

Arguments

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

`TFactors` A vector of time factors corresponding to the compounding value. `TFactors` must be equal to or greater than zero.

`Compounding` (Optional) Scalar value representing the rate at which the input zero rates were compounded when annualized. Default = `2`. This argument determines the formula for the discount factors:

`Compounding` = `1, 2, 3, 4, 6, 12`

`Disc = (1 + Z/F)^(-T)`, where `F` is the compounding frequency, `Z` is the zero rate, and `T` is the time in periodic units, e.g. `T = F` is one year.

`Compounding` = `365`

`Disc = (1 + Z/F)^(-T)`, where `F` is the number of days in the basis year and `T` is a number of days elapsed computed by basis.

`Compounding` = `-1`

`Disc = exp(-T*Z)`, where `T` is time in years.

`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. 0 = actual/actual (default), 1 = 30/360 (SIA), 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).
`EndMonthRule`	(Optional) End-of-month rule. A vector. This rule applies only when `Maturity` is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond's coupon payment date is always the same numerical day of the month. 1 = set rule on (default), meaning that a bond's coupon payment date is always the last actual day of the month.

Description

`Dates = time2date(Settle, TFactors, Compounding, Basis, EndMonthRule)` computes dates corresponding to the times occurring beyond the settlement date.

The `time2date` function is the inverse of `date2time`.

Examples

Show that `date2time` and `time2date` are the inverse of each other. First compute the time factors using `date2time`.

```
Settle = '1-Sep-2002';
Dates = datenum(['31-Aug-2005'; '28-Feb-2006'; '15-Jun-2006';
                 '31-Dec-2006']);
Compounding = 2;
Basis = 0;
EndMonthRule = 1;
TFactors = date2time(Settle, Dates, Compounding, Basis,...
                  EndMonthRule)

TFactors =

    5.9945
    6.9945
    7.5738
    8.6576
```

Now use the calculated `TFactors` in `time2date` and compare the calculated dates with the original set.

```
Dates_calc = time2date(Settle, TFactors, Compounding, Basis,...
                       EndMonthRule)

Dates_calc =

        732555
        732736
        732843
        733042

datestr(Dates_calc)

ans =

31-Aug-2005
28-Feb-2006
15-Jun-2006
31-Dec-2006
```

See Also

`cftimes`, `date2time`

Purpose Current date

Syntax

```
Datenum = today
```

Description `Datenum = today` returns the current date as a serial date number.

Examples

```
Datenum = today
```

returns

```
Datenum =

     730695
```

on July 28, 2000.

See Also `datenum`, `datestr`, `now`

tr2bonds

Purpose Term-structure parameters given Treasury bond parameters

Syntax

```
[Bonds, Prices, Yields] = tr2bonds(TreasuryMatrix, Settle)
```

Arguments

`TreasuryMatrix`	Treasury bond parameters. An n-by-5 matrix, where each row describes a Treasury bond. Columns are `[CouponRate Maturity Bid Asked AskYield]` where

`CouponRate`	Coupon rate, as a decimal fraction.
`Maturity`	Maturity date, as a serial date number. Use `datenum` to convert date strings to serial date numbers.
`Bid`	Bid price based on $100 face value.
`Asked`	Asked price based on $100 face value.
`AskYield`	Asked yield to maturity, as a decimal fraction.

`Settle`	(Optional) Date string or serial date number of the settlement date for the analysis.

Description `[Bonds, Prices, Yields] = tr2bonds(TreasuryMatrix, Settle)` returns term-structure parameters (bond information, prices, and yields) sorted by ascending maturity date, given Treasury bond parameters. The formats of the output matrix and vectors meet requirements for input to the `zbtprice` and `zbtyield` zero-curve bootstrapping functions.

`Bonds`	Coupon bond information. An n-by-6 matrix where each row describes a bond. Columns are `[Maturity CouponRate Face Period Basis EndMonthRule]` where:

`Maturity`	Maturity date of the bond, as a serial date number. Use `datestr` to convert serial date numbers to date strings.
`CouponRate`	Coupon rate of the bond, as a decimal fraction.
`Face`	Redemption or face value of the bond, always 100.
`Period`	Coupons per year of the bond, always 2.
`Basis`	Day-count basis of the bond, always 0 (actual/actual).
`EndMonthRule`	End-of-month flag, always 1, meaning that a bond's coupon payment date is always the last day of the month.

`Prices`	Prices. A column vector containing the price of each bond in `bonds`, respectively. The number of rows (n) matches the number of rows in `bonds`.
`Yields`	Yields. A column vector containing the yield to maturity of each bond in `bonds`, respectively. The number of rows (n) matches the number of rows in `bonds`. If `Settle` is input, `Yields` is computed as a semiannual yield to maturity. If `Settle` is not input, the quoted input yields will be used.

Examples

Given published Treasury bond market parameters for December 22, 1997

```
Matrix =[0.0650 datenum('15-apr-1999')  101.03125 101.09375 0.0564
          0.05125 datenum('17-dec-1998')  99.4375   99.5     0.0563
          0.0625 datenum('30-jul-1998')  100.3125  100.375   0.0560
        0.06125 datenum('26-mar-1998') 100.09375 100.15625 0.0546];
```

Execute the function.

```
[Bonds, Prices, Yields] = tr2bonds(Matrix)
```

```
Bonds =

  729840    0.06125    100    2    0    1
  729966    0.0625     100    2    0    1
  730106    0.05125    100    2    0    1
  730225    0.065      100    2    0    1

Prices =

         100.1563
         100.3750
          99.5000
         101.0938

Yields =

         0.0546
         0.056
         0.0563
         0.0564
```

(Example output has been formatted for readability.)

See Also

`tbl2bond`, `zbtprice`, `zbtyield`, and other functions for Term Structure of Interest Rates

Purpose

Univariate GARCH(P,Q) parameter estimation with Gaussian innovations

Syntax

```
[Kappa, Alpha, Beta] = ugarch(U, P, Q)
```

Arguments

`U`	Single column vector of random disturbances, i.e., the residuals or innovations (ε_t), of an econometric model representing a mean-zero, discrete-time stochastic process. The innovations time series `U` is assumed to follow a GARCH(P,Q) process.
`P`	Non-negative, scalar integer representing a model order of the GARCH process. `P` is the number of lags of the conditional variance. `P` can be zero; when `P = 0`, a GARCH(0,Q) process is actually an ARCH(Q) process.
`Q`	Positive, scalar integer representing a model order of the GARCH process. `Q` is the number of lags of the squared innovations.

Description

`[Kappa, Alpha, Beta] = ugarch(U, P, Q)` computes estimated univariate GARCH(P,Q) parameters with Gaussian innovations.

`Kappa` is the estimated scalar constant term (κ) of the GARCH process.

`Alpha` is a `P`-by-1 vector of estimated coefficients, where `P` is the number of lags of the conditional variance included in the GARCH process.

`Beta` is a `Q`-by-1 vector of estimated coefficients, where `Q` is the number of lags of the squared innovations included in the GARCH process.

The time-conditional variance, σ_t^2, of a GARCH(P,Q) process is modeled as

$$\sigma_t^2 = \kappa + \sum_{i=1}^{P} \alpha_i \sigma_{t-i}^2 + \sum_{j=1}^{Q} \beta_j \varepsilon_{t-j}^2$$

where α represents the argument `Alpha`, β represents `Beta`, and the GARCH(P, Q) coefficients $\{\kappa, \alpha, \beta\}$ are subject to the following constraints.

$$\sum_{i=1}^{P} a_i + \sum_{j=1}^{Q} \beta_j < 1$$

$$\kappa > 0$$

$$a_i \geq 0 \qquad i = 1, 2, \ldots, P$$

$$\beta_j \geq 0 \qquad j = 1, 2, \ldots, Q$$

Note that `U` is a vector of residuals or innovations (ε_t) of an econometric model, representing a mean-zero, discrete-time stochastic process.

Although σ_t^2 is generated using the equation above, ε_t and σ_t^2 are related as

$$\varepsilon_t = \sigma_t v_t$$

where $\{v_t\}$ is an independent, identically distributed (i.i.d.) sequence ~ N(0,1).

Note `ugarch` corresponds generally to the GARCH Toolbox function `garchfit`. The GARCH Toolbox provides a comprehensive and integrated computing environment for the analysis of volatility in time series. For information, see the *GARCH Toolbox User's Guide* or the financial products Web page at `http://www.mathworks.com/products/finprod/`.

Examples See `ugarchsim` for an example of a GARCH(P,Q) process.

See Also `ugarchpred`, `ugarchsim`, and the GARCH Toolbox function `garchfit`

References James D. Hamilton, *Time Series Analysis*, Princeton University Press, 1994

Purpose

Log-likelihood objective function of univariate GARCH(P,Q) processes with Gaussian innovations

Syntax

```
LogLikelihood = ugarchllf(Parameters, U, P, Q)
```

Arguments

`Parameters`	`(1 + P + Q)`- by-1 column vector of GARCH(P,Q) process parameters. The first element is the scalar constant term κ of the GARCH process; the next `P` elements are coefficients associated with the `P` lags of the conditional variance terms; the next `Q` elements are coefficients associated with the `Q` lags of the squared innovations terms.
`U`	Single column vector of random disturbances, i.e., the residuals or innovations (ε_t), of an econometric model representing a mean-zero, discrete-time stochastic process. The innovations time series `U` is assumed to follow a GARCH(P,Q) process.
`P`	Nonnegative, scalar integer representing a model order of the GARCH process. `P` is the number of lags of the conditional variance. `P` can be zero; when `P = 0`, a GARCH(0,Q) process is actually an ARCH(Q) process.
`Q`	Positive, scalar integer representing a model order of the GARCH process. `Q` is the number of lags of the squared innovations.

Description

`LogLikelihood = ugarchllf(Parameters, U, P, Q)` computes the log-likelihood objective function of univariate GARCH(P,Q) processes with Gaussian innovations.

`LogLikelihood` is a scalar value of the GARCH(P,Q) log-likelihood objective function given the input arguments. This function is meant to be optimized via the `fmincon` function of the Optimization Toolbox.

`fmincon` is a minimization routine. To maximize the log-likelihood function, the `LogLikelihood` output parameter is actually the negative of what is formally presented in most time series or econometrics references.

The time-conditional variance, σ_t^2, of a GARCH(P,Q) process is modeled as

$$\sigma_t^2 = \kappa + \sum_{i=1}^{P} \alpha_i \sigma_{t-i}^2 + \sum_{j=1}^{Q} \beta_j \varepsilon_{t-j}^2$$

where α represents the argument `Alpha`, and β represents `Beta`.

`U` is a vector of residuals or innovations (ε_t) representing a mean-zero, discrete time stochastic process. Although σ_t^2 is generated via the equation above, ε_t and σ_t^2 are related as

$$\varepsilon_t = \sigma_t v_t$$

where $\{v_t\}$ is an independent, identically distributed (i.i.d.) sequence ~ N(0,1).

Since `ugarchllf` is really just a helper function, no argument checking is performed. This function is not meant to be called directly from the command line.

Note The GARCH Toolbox provides a comprehensive and integrated computing environment for the analysis of volatility in time series. For information see the *GARCH Toolbox User's Guide* or the financial products Web page at `http://www.mathworks.com/products/finprod/`.

See Also

`ugarch`, `ugarchpred`, `ugarchsim`

Purpose

Forecast conditional variance of univariate GARCH(P,Q) processes

Syntax

```
[VarianceForecast, H] = ugarchpred(U, Kappa, Alpha, Beta,
  NumPeriods)
```

Arguments

`U`	Single column vector of random disturbances, i.e., the residuals or innovations (ε_t), of an econometric model representing a mean-zero, discrete-time stochastic process. The innovations time series `U` is assumed to follow a GARCH(P,Q) process.
`Kappa`	Scalar constant term κ of the GARCH process.
`Alpha`	`P`-by-1 vector of coefficients, where `P` is the number of lags of the conditional variance included in the GARCH process. `Alpha` can be an empty matrix, in which case `P` is assumed 0; when `P = 0`, a GARCH(0,Q) process is actually an ARCH(Q) process.
`Beta`	`Q`-by-1 vector of coefficients, where `Q` is the number of lags of the squared innovations included in the GARCH process.
`NumPeriods`	Positive, scalar integer representing the forecast horizon of interest, expressed in periods compatible with the sampling frequency of the input innovations column vector `U`.

Description

`[VarianceForecast, H] = ugarchpred(U, Kappa, Alpha, Beta, NumPeriods)` forecasts the conditional variance of univariate GARCH(P,Q) processes.

`VarianceForecast` is a number of periods (`NUMPERIODS`)-by-1 vector of the minimum mean-square error forecast of the conditional variance of the innovations time series vector `U` (i.e., ε_t). The first element contains the 1-period-ahead forecast, the second element contains the 2-period-ahead forecast, and so on. Thus, if a forecast horizon greater than 1 is specified (`NUMPERIODS > 1`), the forecasts of all intermediate horizons are returned as well. In this case, the last element contains the variance forecast of the specified horizon, `NumPeriods` from the most recent observation in `U`.

`H` is a vector of the conditional variances (σ_t^2) corresponding to the innovations vector `U`. It is inferred from the innovations `U`, and is a reconstruction of the "past" conditional variances, whereas the `VarianceForecast` output represents the projection of conditional variances into the "future." This sequence is based on setting pre-sample values of σ_t^2 to the unconditional variance of the $\{\varepsilon_t\}$ process. `H` is a single column vector of the same length as the input innovations vector `U`.

The time-conditional variance, σ_t^2, of a GARCH(P,Q) process is modeled as

$$\sigma_t^2 = \kappa + \sum_{i=1}^{P} \alpha_i \sigma_{t-i}^2 + \sum_{j=1}^{Q} \beta_j \varepsilon_{t-j}^2$$

where α represents the argument `Alpha`, β represents `Beta`, and the GARCH(P,Q) coefficients $\{\kappa, \alpha, \beta\}$ are subject to the following constraints.

$$\sum_{i=1}^{P} a_i + \sum_{j=1}^{Q} \beta_j < 1$$

$$\begin{aligned} \kappa &> 0 & & \\ a_i &\geq 0 & i &= 1, 2, \ldots, P \\ \beta_j &\geq 0 & j &= 1, 2, \ldots, Q \end{aligned}$$

Note that `U` is a vector of residuals or innovations (ε_t) of an econometric model, representing a mean-zero, discrete-time stochastic process.

Although σ_t^2 is generated using the equation above, ε_t and σ_t^2 are related as

$$\varepsilon_t = \sigma_t v_t$$

where $\{v_t\}$ is an independent, identically distributed (i.i.d.) sequence ~ N(0,1).

Note `ugarchpred` corresponds generally to the GARCH Toolbox function `garchpred`. The GARCH Toolbox provides a comprehensive and integrated computing environment for the analysis of volatility in time series. For information see the *GARCH Toolbox User's Guide* or the financial products Web page at `http://www.mathworks.com/products/finprod/`.

Examples See `ugarchsim` for an example of forecasting the conditional variance of a univariate GARCH(P,Q) process.

See Also `ugarch`, `ugarchsim`, and the GARCH Toolbox function `garchpred`

ugarchsim

Purpose Simulate a univariate GARCH(P,Q) process with Gaussian innovations

Syntax

```
[U, H] = ugarchsim(Kappa, Alpha, Beta, NumSamples)
```

Arguments

`Kappa`	Scalar constant term κ of the GARCH process.
`Alpha`	`P`-by-1 vector of coefficients, where `P` is the number of lags of the conditional variance included in the GARCH process. `Alpha` can be an empty matrix, in which case `P` is assumed 0; when `P = 0`, a GARCH(0,Q) process is actually an ARCH(Q) process.
`Beta`	`Q`-by-1 vector of coefficients, where `Q` is the number of lags of the squared innovations included in the GARCH process.
`NumSamples`	Positive, scalar integer indicating the number of samples of the innovations `U` and conditional variance `H` (see below) to simulate.

Description

`[U, H] = ugarchsim(Kappa, Alpha, Beta, NumSamples)` simulates a univariate GARCH(P,Q) process with Gaussian innovations.

`U` is a number of samples (`NUMSAMPLES`)-by-1 vector of innovations (ε_t), representing a mean-zero, discrete-time stochastic process. The innovations time series `U` is designed to follow the GARCH(P,Q) process specified by the inputs `Kappa`, `Alpha`, and `Beta`.

`H` is a `NUMSAMPLES`-by-1 vector of the conditional variances (σ_t^2) corresponding to the innovations vector `U`. Note that `U` and `H` are the same length, and form a "matching" pair of vectors. As shown in the following equation, σ_t^2 (i.e., `H(t)`) represents the time series inferred from the innovations time series $\{\varepsilon_t\}$ (i.e., `U`).

The time-conditional variance, σ_t^2, of a GARCH(P,Q) process is modeled as

$$\sigma_t^2 = \kappa + \sum_{i=1}^{P} \alpha_i \sigma_{t-i}^2 + \sum_{j=1}^{Q} \beta_j \varepsilon_{t-j}^2$$

where α represents the argument `Alpha`, β represents `Beta`, and the GARCH(P,Q) coefficients $\{\kappa, \alpha, \beta\}$ are subject to the following constraints.

$$\sum_{i=1}^{P} a_i + \sum_{j=1}^{Q} \beta_j < 1$$

$$\begin{aligned} \kappa &> 0 \\ a_i &\geq 0 \qquad i = 1, 2, \ldots, P \\ \beta_j &\geq 0 \qquad j = 1, 2, \ldots, Q \end{aligned}$$

Note that `U` is a vector of residuals or innovations (ε_t) of an econometric model, representing a mean-zero, discrete-time stochastic process.

Although σ_t^2 is generated using the equation above, ε_t and σ_t^2 are related as

$$\varepsilon_t = \sigma_t v_t$$

where $\{v_t\}$ is an independent, identically distributed (i.i.d.) sequence ~ N(0,1).

The output vectors `U` and `H` are designed to be steady-state sequences in which transients have arbitrarily small effect. The (arbitrary) metric used by `ugarchsim` strips the first `N` samples of `U` and `H` such that the sum of the GARCH coefficients, excluding `Kappa`, raised to the `N`th power, does not exceed 0.01.

```
0.01 = (sum(Alpha) + sum(Beta))^N
```

Thus

```
N = log(0.01)/log((sum(Alpha) + sum(Beta)))
```

Note `ugarchsim` corresponds generally to the GARCH Toolbox function `garchsim`. The GARCH Toolbox provides a comprehensive and integrated computing environment for the analysis of volatility in time series. For information see the *GARCH Toolbox User's Guide* or the financial products Web page at `http://www.mathworks.com/products/finprod/`.

Examples

This example simulates a GARCH(P,Q) process with `P = 2` and `Q = 1`.

```
% Set the random number generator seed for reproducability.

randn('seed', 10)

% Set the simulation parameters of GARCH(P,Q) = GARCH(2,1) process.

Kappa = 0.25;       %a positive scalar.
Alpha = [0.2 0.1]'; %a column vector of nonnegative numbers (P = 2).
Beta = 0.4;         % Q = 1.
NumSamples = 500;  % number of samples to simulate.

% Now simulate the process.

[U , H] = ugarchsim(Kappa, Alpha, Beta, NumSamples);

% Estimate the process parameters.

P = 2;    % Model order P (P = length of Alpha).
Q = 1;    % Model order Q (Q = length of Beta).
[k, a, b] = ugarch(U , P , Q);
disp(' ')
disp(' Estimated Coefficients:')
disp(' ----------------------')
disp([k; a; b])
disp(' ')

% Forecast the conditional variance using the estimated
% coefficients.

NumPeriods = 10;    % Forecast out to 10 periods.
[VarianceForecast, H1] = ugarchpred(U, k, a, b, NumPeriods);
disp(' Variance Forecasts:')
disp(' ------------------')
disp(VarianceForecast)
disp(' ')
```

When the above code is executed, the screen output looks like the display shown.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Diagnostic Information

Number of variables: 4

Functions
Objective:        ugarchllf
Gradient:         finite-differencing
Hessian:          finite-differencing (or Quasi-Newton)

Constraints
Nonlinear constraints:            do not exist
Number of linear inequality constraints:    1
Number of linear equality constraints:      0
Number of lower bound constraints:          4
Number of upper bound constraints:          0
Algorithm selected
   medium-scale

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
End diagnostic information
                               max                       Directional
 Iter   F-count     f(x)    constraint   Step-size        derivative     Procedure
    1       5     699.185      -0.125            1       -2.97e+006
    2      22     658.224     -0.1249     0.000488            -64.6
    3      28     610.181           0            1            -49.4
    4      35     590.888           0          0.5            -38.9
    5      42     583.961    -0.03317          0.5            -29.8
    6      49     583.224    -0.02756          0.5            -31.8
    7      57     582.947    -0.02067         0.25            -7.28
    8      63     578.182           0            1            -2.43
    9      71     578.138    -0.09145         0.25            -0.55
   10      77     577.898    -0.04452            1           -0.148
   11      84     577.882    -0.06128          0.5          -0.0488
   12      90     577.859    -0.07117            1        -0.000758
   13      96     577.858    -0.07033            1        -0.000305    Hessian modified
   14     102     577.858    -0.07042            1       -3.32e-005    Hessian modified
   15     108     577.858     -0.0707            1       -1.29e-006    Hessian modified
   16     114     577.858    -0.07077            1       -1.29e-007    Hessian modified
   17     120     577.858    -0.07081            1       -1.97e-007    Hessian modified
```

```
Optimization Converged Successfully
Magnitude of directional derivative in search direction
  less than 2*options.TolFun and maximum constraint violation
  is less than options.TolCon
No Active Constraints

Estimated Coefficients:
----------------------
0.2520
0.0708
0.1623
0.4000

Variance Forecasts:
------------------
1.3243
0.9594
0.9186
0.8402
0.7966
0.7634
0.7407
0.7246
0.7133
0.7054
```

See Also

`ugarch`, `ugarchpred`, and the GARCH Toolbox function `garchsim`

References

James D. Hamilton, *Time Series Analysis*, Princeton University Press, 1994

Purpose Day of the week

Syntax

```
[DayNum, DayString] = weekday(Date)
```

Description `[DayNum, DayString] = weekday(Date)` returns the day of the week in numeric and string form given the date as a serial date number or date string. The days of the week have these values.

DayNum	DayString
1	Sun
2	Mon
3	Tue
4	Wed
5	Thu
6	Fri
7	Sat

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

Examples

```
[DayNum, DayString] = weekday(730845)
```

or

```
[DayNum, DayString] = weekday('25-Dec-2000')
```

returns

```
DayNum =

     2

DayString =

Mon
```

See Also

datenum, datestr, datevec, day

Purpose Number of working days between dates

Syntax

```
Days = wrkdydif(StartDate, EndDate, Holidays)
```

Description `Days = wrkdydif(StartDate, EndDate, Holidays)` returns the number of working days between dates `StartDate` and `EndDate`. `Holidays` is the number of holidays between the given dates, an integer. Enter dates as serial date numbers or date strings.

Examples

```
Days = wrkdydif('9/1/2000', '9/11/2000', 1)
```

or

```
Days = wrkdydif(730730, 730740, 1)
```

returns

```
Days =

        6
```

See Also `busdate`, `datewrkdy`, `days360`, `days365`, `daysact`, `daysdif`, `holidays`, `yearfrac`

x2mdate

Purpose

Excel serial date number to MATLAB serial date number

Syntax

```
MATLABDate = x2mdate(ExcelDateNumber, Convention)
```

Arguments

`ExcelDateNumber`	A vector or scalar of Excel serial date numbers.
`Convention`	(Optional) Excel date system. A vector or scalar. When `Convention = 0` (default), the Excel 1900 date system is in effect. When `Convention = 1`, the Excel 1904 date system in used. In the Excel 1900 date system, the Excel serial date number 1 corresponds to January 1, 1900 A.D. In the Excel 1904 date system, date number 0 is January 1, 1904 A.D.

Vector arguments must have consistent dimensions.

Description

`DateNumber = x2mdate(ExcelDateNumber, Convention)` converts Excel serial date numbers to MATLAB serial date numbers. MATLAB date numbers start with 1 = January 1, 0000 A.D., hence there is a difference of 693961 relative to the 1900 date system, or 695422 relative to the 1904 date system. This function is useful with MATLAB Excel Link.

Examples

Given Excel date numbers in the 1904 system

```
ExDates = [35423  35788  36153];
```

convert them to MATLAB date numbers

```
MATLABDate = x2mdate(ExDates, 1)

MATLABDate =

       730845       731210       731575
```

and then to date strings.

```
datestr(MATLABDate)

ans =

25-Dec-2000
25-Dec-2001
25-Dec-2002
```

See Also `datenum`, `datestr`, `m2xdate`

xirr

Purpose

Internal rate of return for nonperiodic cash flow

Syntax

```
Return = xirr(CashFlow, CashFlowDates, Guess, MaxIterations)
```

Arguments

`CashFlow`	A vector of nonperiodic cash flows. Include the initial investment as the initial cash flow value (a negative number).
`CashFlowDates`	A vector of dates on which the cash flows occur. Enter dates as serial date numbers or date strings.
`Guess`	(Optional) Initial estimate of the expected return. Default = 0.1 (10%).
`MaxIterations`	(Optional) Number of iterations used by Newton's method to solve for `Return`. Default = 50.

Description

`Return = xirr(CashFlow, CashFlowDates, Guess, MaxIterations)` returns the internal rate of return for a schedule of nonperiodic cash flows.

Examples

An investment of $10,000 returns this nonperiodic cash flow. The original investment and its date are included.

Cash flow	Dates
($10000)	January 12, 2000
$2500	February 14, 2001
$2000	March 3, 2001
$3000	June 14, 2001
$4000	December 1, 2001

To calculate the internal rate of return for this nonperiodic cash flow

```
CashFlow = [-10000, 2500, 2000, 3000, 4000];
CashFlowDates = ['01/12/2000'
                 '02/14/2001'
                 '03/03/2001'
                 '06/14/2001'
                 '12/01/2001'];
```

```
Return = xirr(CashFlow, CashFlowDates)
```

returns

```
Return =
        0.1009
```
(or 10.09%)

See Also `fvvar`, `irr`, `mirr`, `pvvar`

References Sharpe and Alexander, *Investments*, 4th edition, page 463.

year

Purpose Year of date

Syntax

```
Year = year(Date)
```

Description `Year = year(Date)` returns the year of a serial date number or a date string.

Examples

```
Year = year(731798.776)
```

or

```
Year = year('05-Aug-2003')
```

returns

```
Year =

        2003
```

See Also `datevec`, `day`, `month`, `yeardays`

yeardays

Purpose Number of days in year

Syntax

```
Days = yeardays(Year, Basis)
```

Arguments

`Year`	Enter as a four-digit integer.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. 0 = actual/actual (default), 1 = 30/360 (SIA), 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).

Description `Days = yeardays(Year, Basis)` returns the number of days in the given year, based upon the day-count basis.

Examples

```
Days = yeardays(2000)

Days =

       366

Days = yeardays(2000, 1)

Days =

   360
```

See Also `days360`, `days365`, `daysact`, `year`, `yearfrac`

yearfrac

Purpose Fraction of year between dates

Syntax

```
Fraction = yearfrac(StartDate, EndDate, Basis)
```

Arguments

`StartDate`	Enter as serial date numbers or date strings.
`EndDate`	Enter as serial date numbers or date strings.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. 0 = actual/actual (default), 1 = 30/360 (SIA), 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).

All specified arguments must be number of instruments (`NUMINST`) by 1 or 1-by-`NUMINST` conforming vectors or scalar arguments.

Description

`Fraction = yearfrac(StartDate, EndDate, Basis)` returns a fraction based on the number of days between dates `StartDate` and `EndDate` using the given day-count basis. If `EndDate` is earlier than `StartDate`, `Fraction` is negative.

Examples

```
Fraction = yearfrac('14 mar 01', '14 sep 01', 0)

Fraction =

    0.5041

Fraction = yearfrac('14 mar 01', '14 sep 01', 1)

Fraction =

    0.5000
```

See Also

`days360`, `days365`, `daysact`, `daysdif`, `months`, `wrkdydif`, `yeardays`

Purpose

Yield of discounted security

Syntax

```
Yield = ylddisc(Settle, Maturity, Face, Price, Basis)
```

Arguments

`Settle`	Settlement date. Enter as serial date number or date string. `Settle` must be earlier than or equal to `Maturity`.
`Maturity`	Maturity date. Enter as serial date number or date string.
`Face`	Redemption (par, face) value.
`Price`	Discounted price of the security.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. 0 = actual/actual (default), 1 = 30/360 (SIA), 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).

Description

`Yield = ylddisc(Settle, Maturity, Face, Price, Basis)` finds the yield of a discounted security.

Examples

Using the data

```
Settle = '10/14/2000';
Maturity = '03/17/2001';
Face = 100;
Price = 96.28;
Basis = 2;

Yield = ylddisc(Settle, Maturity, Face, Price, Basis)
```

returns

```
Yield =

          0.0903  (or 9.03%)
```

See Also

`acrudisc`, `bndprice`, `bndyield`, `prdisc`, `yldmat`, `yldtbill`

References

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

yldmat

Purpose Yield with interest at maturity

Syntax

```
Yield = yldmat(Settle, Maturity, Issue, Face, Price, CouponRate,
   Basis)
```

Arguments

`Settle`	Settlement date. Enter as serial date number or date string. `Settle` must be earlier than or equal to `Maturity`.
`Maturity`	Maturity date. Enter as serial date number or date string.
`Issue`	Issue date. Enter as serial date number or date string.
`Face`	Redemption (par, face) value.
`Price`	Price of the security.
`CouponRate`	Coupon rate. Enter as decimal fraction.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. 0 = actual/actual (default), 1 = 30/360 (SIA), 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).

Description `Yield = yldmat(Settle, Maturity, Issue, Face, Price, CouponRate, Basis)` returns the yield of a security paying interest at maturity.

Examples Using the data

```
Settle = '02/07/2000';
Maturity = '04/13/2000';
Issue = '10/11/1999';
Face = 100;
Price = 99.98;
CouponRate = 0.0608;
Basis = 1;

Yield = yldmat(Settle, Maturity, Issue, Face, Price,...
CouponRate, Basis)
```

returns

```
Yield =
```

`0.0607` (or 6.07%)

See Also

`acrubond`, `bndprice`, `bndyield`, `prmat`, `ylddisc`, `yldtbill`

References

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

yldtbill

Purpose

Yield of Treasury bill

Syntax

```
Yield = yldtbill(Settle, Maturity, Face, Price)
```

Arguments

`Settle`	Settlement date. Enter as serial date number or date string. `Settle` must be earlier than or equal to `Maturity`.
`Maturity`	Maturity date. Enter as serial date number or date string.
`Face`	Redemption (par, face) value.
`Price`	Price of the Treasury bill.

Description

`Yield = yldtbill(Settle, Maturity, Face, Price)` returns the yield for a Treasury bill.

Examples

The settlement date of a Treasury bill is February 10, 2000, the maturity date is August 6, 2000, the par value is $1000, and the price is $981.36. Using this data

```
Yield = yldtbill('2/10/2000', '8/6/2000', 1000, 981.36)
```

returns

```
Yield =

        0.0384 (or 3.84%)
```

See Also

`beytbill`, `bndyield`, `prtbill`, `yldmat`

References

Bodie, Kane, and Marcus, *Investments*, pages 41-43.

Purpose

Zero curve bootstrapping from coupon bond data given price

Syntax

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

Arguments

`Bonds`

Coupon bond information used to generate the zero curve. An n-by-2 to n-by-6 matrix where each row describes a bond. The first two columns are required; the rest are optional but must be added in order. All rows in `Bonds` must have the same number of columns.

Columns are `[Maturity CouponRate Face Period Basis EndMonthRule]` where

`Maturity`	Maturity date of the bond, as a serial date number. Use `datenum` to convert date strings to serial date numbers.
`CouponRate`	Coupon rate of the bond, as a decimal fraction.
`Face`	(Optional) Redemption or face value of the bond. Default = 100.
`Period`	(Optional) Coupons per year of the bond, as an integer. Allowed values are 0, 1, 2 (default), 3, 4, 6, and 12.
`Basis`	(Optional) Day-count basis of the bond: 0 = actual/actual (default), 1 = 30/360 (SIA), 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).

`EndMonthRule`	(Optional) End-of-month flag. This flag applies only when `Maturity` is an end-of-month date for a month having 30 or fewer days. 0 = ignore flag, meaning that a bond's coupon payment date is always the same day of the month. 1 = set flag (default), meaning that a bond's coupon payment date is always the last day of the month.

`Prices` A column vector containing the clean price (price without accrued interest) of each bond in `Bonds`, respectively. The number of rows (n) must match the number of rows in `Bonds`.

`Settle` Settlement date, as a scalar serial date number. This represents time zero for deriving the zero curve, and it is normally the common settlement date for all the bonds.

`OutputCompounding` (Optional) A scalar that sets the compounding frequency per year for the output zero rates in `ZeroRates`. Allowed values are:

1	annual compounding
2	semiannual compounding (default)
3	compounding three times per year
4	quarterly compounding
6	bimonthly compounding
12	monthly compounding
-1	continuous compounding

Description

`[ZeroRates, CurveDates] = zbtprice(Bonds, Prices, Settle, OutputCompounding)` uses the bootstrap method to return a zero curve given a portfolio of coupon bonds and their prices. A zero curve consists of the yields to maturity for a portfolio of theoretical zero-coupon bonds that are derived from the input `Bonds` portfolio. The bootstrap method that this function uses does *not* require alignment among the cash-flow dates of the bonds in the input portfolio. It uses theoretical par bond arbitrage and yield interpolation to derive all zero rates. For best results, use a portfolio of at least 30 bonds evenly spaced across the investment horizon.

`ZeroRates` An m-by-1 vector of decimal fractions that are the implied zero rates for each point along the investment horizon represented by `CurveDates`; m is the number of bonds of unique maturity dates. In aggregate, the rates in `ZeroRates` constitute a zero curve.

If more than one bond has the same maturity date, `zbtprice` returns the mean zero rate for that maturity.

`CurveDates` An m-by-1 vector of unique maturity dates (as serial date numbers) that correspond to the zero rates in `ZeroRates`; m is the number of bonds of different maturity dates. These dates begin with the earliest maturity date and end with the latest maturity date `Maturity` in the `Bonds` matrix.

Examples

Given data and prices for 12 coupon bonds, two with the same maturity date; and given the common settlement date

```
Bonds = [datenum('6/1/1998')   0.0475   100  2  0  0;
         datenum('7/1/2000')   0.06     100  2  0  0;
         datenum('7/1/2000')   0.09375  100  6  1  0;
         datenum('6/30/2001')  0.05125  100  1  3  1;
         datenum('4/15/2002')  0.07125  100  4  1  0;
         datenum('1/15/2000')  0.065    100  2  0  0;
         datenum('9/1/1999')   0.08     100  3  3  0;
         datenum('4/30/2001')  0.05875  100  2  0  0;
         datenum('11/15/1999') 0.07125  100  2  0  0;
         datenum('6/30/2000')  0.07     100  2  3  1;
         datenum('7/1/2001')   0.0525   100  2  3  0;
         datenum('4/30/2002')  0.07     100  2  0  0];

Prices = [99.375;
          99.875;
         105.75 ;
          96.875;
         103.625;
```

```
            101.125;
            103.125;
             99.375;
            101.0  ;
            101.25 ;
             96.375;
            102.75 ];

Settle = datenum('12/18/1997');
```

Set semiannual compounding for the zero curve.

```
OutputCompounding = 2;
```

Execute the function

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

which returns the zero curve at the maturity dates. Note the mean zero rate for the two bonds with the same maturity date*.

```
ZeroRates =

               0.0616
               0.0609
               0.0658
               0.0590
               0.0648
               0.0655*
               0.0606
               0.0601
               0.0642
               0.0621
               0.0627

CurveDates =

                729907    (serial date number for 01-Jun-1998)
                730364    (01-Sep-1999)
                730439    (15-Nov-1999)
                730500    (15-Jan-2000)
```

`730667`	(30-Jun-2000)
`730668`	(01-Jul-2000)*
`730971`	(30-Apr-2001)
`731032`	(30-Jun-2001)
`731033`	(01-Jul-2001)
`731321`	(15-Apr-2002)
`731336`	(30-Apr-2002)

See Also

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

References

Fabozzi, Frank J. “The Structure of Interest Rates.” Ch. 6 in Fabozzi, Frank J. and T. Dessa Fabozzi, eds. *The Handbook of Fixed Income Securities.* 4th ed. New York: Irwin Professional Publishing. 1995.

McEnally, Richard W. and James V. Jordan. “The Term Structure of Interest Rates.” Ch. 37 in Fabozzi and Fabozzi, ibid.

Das, Satyajit. “Calculating Zero Coupon Rates.” *Swap and Derivative Financing*. Appendix to Ch. 8, pp. 219-225. New York: Irwin Professional Publishing. 1994.

zbtyield

Purpose

Zero curve bootstrapping from coupon bond data given yield

Syntax

```
[ZeroRates, CurveDates] = zbtyield(Bonds, Yields, Settle,
   OutputCompounding)
```

Arguments

`Bonds`	Coupon bond information used to generate the zero curve. An n-by-2 to n-by-6 matrix where each row describes a bond. The first two columns are required; the rest are optional but must be added in order. All rows in `Bonds` must have the same number of columns. Columns are `[Maturity CouponRate Face Period Basis EndMonthRule]` where

`Maturity`	Maturity date of the bond, as a serial date number. Use `datenum` to convert date strings to serial date numbers.
`CouponRate`	Coupon rate of the bond, as a decimal fraction.
`Face`	(Optional) Redemption or face value of the bond. Default = 100.
`Period`	(Optional) Coupons per year of the bond, as an integer. Allowed values are 0, 1, 2 (default), 3, 4, 6, and 12.
`Basis`	(Optional) Day-count basis of the bond. 0 = actual/actual (default), 1 = 30/360 (SIA), 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).

`EndMonthRule`	(Optional) End-of-month flag. This flag applies only when `Maturity` is an end-of-month date for a month having 30 or fewer days. 0 = ignore flag, meaning that a bond's coupon payment date is always the same day of the month. 1 = set flag (default), meaning that a bond's coupon payment date is always the last day of the month.

`Yields`	A column vector containing the yield to maturity of each bond in `Bonds`, respectively. The number of rows (n) must match the number of rows in `Bonds`.
`Settle`	Settlement date, as a scalar serial date number. This represents time zero for deriving the zero curve, and it is normally the common settlement date for all the bonds.
`OutputCompounding`	(Optional) A scalar that sets the compounding frequency per year for the output zero rates in `ZeroRates`. Allowed values are:

1	annual compounding
2	semiannual compounding (default)
3	compounding three times per year
4	quarterly compounding
6	bimonthly compounding
12	monthly compounding
-1	continuous compounding

Description

`[ZeroRates, CurveDates] = zbtyield(Bonds, Yields, Settle, OutputCompounding)` uses the bootstrap method to return a zero curve given a portfolio of coupon bonds and their yields. A zero curve consists of the yields to maturity for a portfolio of theoretical zero-coupon bonds that are derived from the input `Bonds` portfolio. The bootstrap method that this function uses does *not* require alignment among the cash-flow dates of the bonds in the input portfolio. It uses theoretical par bond arbitrage and yield interpolation to derive all zero rates. For best results, use a portfolio of at least 30 bonds evenly spaced across the investment horizon.

`ZeroRates` An m-by-1 vector of decimal fractions that are the implied zero rates for each point along the investment horizon represented by `CurveDates`; m is the number of bonds of different maturity dates. In aggregate, the rates in `ZeroRates` constitute a zero curve.

If more than one bond has the same maturity date, `zbtyield` returns the mean zero rate for that maturity.

`CurveDates` An m-by-1 vector of unique maturity dates (as serial date numbers) that correspond to the zero rates in `ZeroRates`; m is the number of bonds of different maturity dates. These dates begin with the earliest maturity date and end with the latest maturity date `Maturity` in the `Bonds` matrix. Use `datestr` to convert serial date numbers to date strings.

Examples

Given data and yields to maturity for 12 coupon bonds, two with the same maturity date; and given the common settlement date

```
Bonds = [datenum('6/1/1998')   0.0475   100  2  0  0;
         datenum('7/1/2000')   0.06     100  2  0  0;
         datenum('7/1/2000')   0.09375  100  6  1  0;
         datenum('6/30/2001')  0.05125  100  1  3  1;
         datenum('4/15/2002')  0.07125  100  4  1  0;
         datenum('1/15/2000')  0.065    100  2  0  0;
         datenum('9/1/1999')   0.08     100  3  3  0;
         datenum('4/30/2001')  0.05875  100  2  0  0;
         datenum('11/15/1999') 0.07125  100  2  0  0;
         datenum('6/30/2000')  0.07     100  2  3  1;
         datenum('7/1/2001')   0.0525   100  2  3  0;
         datenum('4/30/2002')  0.07     100  2  0  0];

Yields = [0.0616
          0.0605
          0.0687
          0.0612
          0.0615
```

```
            0.0591
            0.0603
            0.0608
            0.0655
            0.0646
            0.0641
            0.0627];

Settle = datenum('12/18/1997');
```

Set semiannual compounding for the zero curve.

```
OutputCompounding = 2;
```

Execute the function

```
[ZeroRates, CurveDates] = zbtyield(Bonds, Yields, Settle,...
OutputCompounding)
```

which returns the zero curve at the maturity dates. Note the mean zero rate for the two bonds with the same maturity date*.

```
ZeroRates =

    0.0616
    0.0575
    0.0692
    0.0613
    0.0616
    0.0596*
    0.0606
    0.0659
    0.0650
    0.0607
    0.0628

CurveDates =

             729907   (serial date number for 01-Jun-1998)
             730364   (01-Sep-1999)
             730439   (15-Nov-1999)
             730500   (15-Jan-2000)
```

730667	(30-Jun-2000)
730668	(01-Jul-2000)*
730971	(30-Apr-2001)
731032	(30-Jun-2001)
731033	(01-Jul-2001)
731321	(15-Apr-2002)
731336	(30-Apr-2002)

See Also

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

References

Fabozzi, Frank J. "The Structure of Interest Rates." Ch. 6 in Fabozzi, Frank J. and T. Dessa Fabozzi, eds. *The Handbook of Fixed Income Securities.* 4th ed. New York: Irwin Professional Publishing. 1995.

McEnally, Richard W. and James V. Jordan. "The Term Structure of Interest Rates." Ch. 37 in Fabozzi and Fabozzi, ibid.

Das, Satyajit. "Calculating Zero Coupon Rates." *Swap and Derivative Financing.* Appendix to Ch. 8, pp. 219-225. New York: Irwin Professional Publishing. 1994.

Purpose Discount curve given a zero curve

Syntax

```
[DiscRates, CurveDates] = zero2disc(ZeroRates, CurveDates, Settle,
   Compounding, Basis)
```

Arguments

`ZeroRates`	A number of bonds (`NUMBONDS`) by 1 vector of annualized zero rates, as decimal fractions. In aggregate, the rates constitute an implied zero curve for the investment horizon represented by `CurveDates`.
`CurveDates`	A `NUMBONDS`-by-1 vector of maturity dates (as serial date numbers) that correspond to the zero rates.
`Settle`	A serial date number that is the common settlement date for the zero rates; i.e., the settlement date for the bonds from which the zero curve was bootstrapped.
`Compounding`	(Optional) A scalar that indicates the compounding frequency per year used for annualizing the input zero rates in `ZeroRates`. Allowed values are: 1 annual compounding 2 semiannual compounding (default) 3 compounding three times per year 4 quarterly compounding 6 bimonthly compounding 12 monthly compounding 365 daily compounding -1 continuous compounding
`Basis`	(Optional) Day-count basis used for annualizing the input zero rates. 0 = actual/actual (default), 1 = 30/360 (SIA), 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).

Description

[DiscRates, CurveDates] = zero2disc(ZeroRates, CurveDates, Settle, Compounding, Basis) returns a discount curve given a zero curve and its maturity dates.

DiscRates	A NUMBONDS-by-1 vector of discount factors, as decimal fractions. In aggregate, the factors in constitute a discount curve for the investment horizon represented by CurveDates.
CurveDates	A NUMBONDS-by-1 vector of maturity dates (as serial date numbers) that correspond to the discount rates. This vector is the same as the input vector CurveDates.

Examples

Given a zero curve over a set of maturity dates and a settlement date

```
ZeroRates = [0.0464
             0.0509
             0.0524
             0.0525
             0.0531
             0.0525
             0.0530
             0.0531
             0.0549
             0.0536];

CurveDates = [datenum('06-Nov-2000')
              datenum('11-Dec-2000')
              datenum('15-Jan-2001')
              datenum('05-Feb-2001')
              datenum('04-Mar-2001')
              datenum('02-Apr-2001')
              datenum('30-Apr-2001')
              datenum('25-Jun-2001')
              datenum('04-Sep-2001')
              datenum('12-Nov-2001')];

Settle = datenum('03-Nov-2000');
```

The zero curve was compounded daily on an actual/365 basis.

```
InputCompounding = 365;
InputBasis = 3;
```

Execute the function

```
[DiscRates, CurveDates] = zero2disc(ZeroRates, CurveDates,...
Settle, Compounding, Basis)
```

which returns the discount curve `DiscRates` at the maturity dates `CurveDates`.

```
DiscRates =

      0.9996
      0.9947
      0.9896
      0.9866
      0.9826
      0.9787
      0.9745
      0.9665
      0.9552
      0.9466

CurveDates =

      730796
      730831
      730866
      730887
      730914
      730943
      730971
      731027
      731098
      731167
```

For readability, `ZeroRates` and `DiscRates` are shown here only to the basis point. However, MATLAB computed them at full precision. If you enter `ZeroRates` as shown, `DiscRates` may differ due to rounding.

See Also

`disc2zero` and other functions for Term Structure of Interest Rates

zero2fwd

Purpose Forward curve given a zero curve

Syntax

```
[ForwardRates, CurveDates] = zero2fwd(ZeroRates, CurveDates,
   Settle, Compounding, Basis
```

Arguments

`ZeroRates` A number of bonds (`NUMBONDS`) by 1 vector of annualized zero rates, as decimal fractions. In aggregate, the rates constitute an implied zero curve for the investment horizon represented by `CurveDates`. The first element pertains to forward rates from the settlement date to the first curve date.

`CurveDates` A `NUMBONDS`-by-1 vector of maturity dates (as serial date numbers) that correspond to the zero rates.

`Settle` A serial date number that is the common settlement date for the zero rates.

`Compounding` (Optional) A scalar that sets the compounding frequency per year used to annualize the input zero rates and the output implied forward rates.Allowed values are:

1	annual compounding
2	semiannual compounding (default)
3	compounding three times per year
4	quarterly compounding
6	bimonthly compounding
12	monthly compounding
365	daily compounding
-1	continuous compounding

`Basis` (Optional) Day-count basis used to construct the input zero and output implied forward rate curves.
0 = actual/actual (default), 1 = 30/360 (SIA),
2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA),
5 = 30/360 (ISDA), 6 = 30/360 (European),
7 = actual/365 (Japanese).

Description

`[ForwardRates, CurveDates] = zero2fwd(ZeroRates, CurveDates, Settle, Compounding, Basis)` returns an implied forward rate curve given a zero curve and its maturity dates.

`ForwardRates`	A `NUMBONDS`-by-1 vector of decimal fractions. In aggregate, the rates in `ForwardRates` constitute a forward curve over the dates in `CurveDates`.
`CurveDates`	A `NUMBONDS`-by-1 vector of maturity dates (as serial date numbers) that correspond to the forward rates. This vector is the same as the input vector `CurveDates`.

Examples

Given a zero curve over a set of maturity dates, a settlement date, and a compounding rate, compute the forward rate curve.

```
ZeroRates = [0.0458
             0.0502
             0.0518
             0.0519
             0.0524
             0.0519
             0.0523
             0.0525
             0.0541
             0.0529];

CurveDates = [datenum('06-Nov-2000')
              datenum('11-Dec-2000')
              datenum('15-Jan-2001')
              datenum('05-Feb-2001')
              datenum('04-Mar-2001')
              datenum('02-Apr-2001')
              datenum('30-Apr-2001')
              datenum('25-Jun-2001')
              datenum('04-Sep-2001')
              datenum('12-Nov-2001')];

Settle = datenum('03-Nov-2000');
Compounding = 1;
```

Execute the function

```
[ForwardRates, CurveDates] = zero2fwd(ZeroRates, CurveDates,...
Settle, Compounding)
```

which returns the forward rate curve `ForwardRates` at the maturity dates `CurveDates`.

```
ForwardRates =

    0.0458
    0.0506
    0.0535
    0.0522
    0.0541
    0.0498
    0.0544
    0.0531
    0.0594
    0.0476

CurveDates =

      730796
      730831
      730866
      730887
      730914
      730943
      730971
      731027
      731098
      731167
```

For readability, `ZeroRates` and `ForwardRates` are shown here only to the basis point. However, MATLAB computed them at full precision. If you enter `ZeroRates` as shown, `ForwardRates` may differ due to rounding.

See Also

`fwd2zero` and other functions for Term Structure of Interest Rates

Purpose

Par yield curve given a zero curve

Syntax

```
[ParRates, CurveDates] = zero2pyld(ZeroRates, CurveDates, Settle,
     Compounding, Basis, OutputCompounding)
```

Arguments

`ZeroRates`	A number of bonds (`NUMBONDS`) by 1 vector of annualized zero rates, as decimal fractions. In aggregate, the rates constitute an implied zero curve for the investment horizon represented by `CurveDates`.
`CurveDates`	A `NUMBONDS`-by-1 vector of maturity dates (as serial date numbers) that correspond to the zero rates.
`Settle`	A serial date number that is the common settlement date for the zero rates.
`Compounding`	(Optional) A scalar that sets the rate at which the implied zero rates are compounded when annualized. Allowed values are: 1 annual compounding 2 semiannual compounding (default) 3 compounding three times per year 4 quarterly compounding 6 bimonthly compounding 12 monthly compounding 365 daily compounding -1 continuous compounding
`Basis`	(Optional) Day-count basis used to annualize the implied zero rates. 0 = actual/actual (default), 1 = 30/360 (SIA), 2 = actual/360, 3 = actual/365, 4 = 30/360 (PSA), 5 = 30/360 (ISDA), 6 = 30/360 (European), 7 = actual/365 (Japanese).
`OutputCompounding`	(Optional) Value representing the rate at which the par rates are compounded. Default = `Compounding`.

Description

`[ParRates, CurveDates] = zero2pyld(ZeroRates, CurveDates, Settle, Compounding, Basis, OutputCompounding)` returns a par yield curve given a zero curve and its maturity dates.

`ParRates`	A `NUMBONDS`-by-1 vector of annualized par yields, as decimal fractions. (Par yields = coupon rates.) In aggregate, the yield rates in `ParRates` constitute a par yield curve for the investment horizon represented by `CurveDates`.
`CurveDates`	A `NUMBONDS`-by-1 vector of maturity dates (as serial date numbers) that correspond to the par yield rates. This vector is the same as the input vector `CurveDates`.

Examples

Given

- A zero curve over a set of maturity dates and
- A settlement date
- Annual compounding for the input zero curve and monthly compounding for the output par rates

compute a par yield curve.

```
ZeroRates = [0.0457
             0.0487
             0.0506
             0.0507
             0.0505
             0.0504
             0.0506
             0.0516
             0.0539
             0.0530];

CurveDates = [datenum('06-Nov-2000')
              datenum('11-Dec-2000')
              datenum('15-Jan-2001')
              datenum('05-Feb-2001')
              datenum('04-Mar-2001')
              datenum('02-Apr-2001')
              datenum('30-Apr-2001')
```

```
              datenum('25-Jun-2001')
              datenum('04-Sep-2001')
              datenum('12-Nov-2001')];

Settle = datenum('03-Nov-2000');
Compounding = 1;
OutputCompounding = 12;

[ParRates, CurveDates] = zero2pyld(ZeroRates, CurveDates,...
Settle, Compounding, [] , OutputCompounding)

ParRates =

    0.0479
    0.0511
    0.0530
    0.0531
    0.0526
    0.0524
    0.0525
    0.0534
    0.0555
    0.0543

CurveDates =

      730796
      730831
      730866
      730887
      730914
      730943
      730971
      731027
      731098
      731167
```

For readability, `ZeroRates` and `ParRates` are shown only to the basis point. However, MATLAB computed them at full precision. If you enter `ZeroRates` as shown, `ParRates` may differ due to rounding.

See Also

`pyld2zero` and other functions for Term Structure of Interest Rates

A

Bibliography

For the well-known algorithms and formulas used in the Financial Toolbox (such as how to compute a loan payment given principal, interest rate, and length of the loan), no references are given here. The references here pertain to less common formulas.

Bond Pricing and Yields

The pricing and yield formulas for fixed-income securities come from:

Mayle, Jan. *Standard Securities Calculation Methods.* New York: Securities Industry Association, Inc. Vol. 1, 3rd ed., 1993, ISBN 1-882936-01-9. Vol. 2, 1994, ISBN 1-882936-02-7.

In many cases these formulas compute the price of a security given yield, dates, rates, and other data. These formulas are nonlinear, however; so when solving for an independent variable within a formula, the Financial Toolbox uses Newton's method. See any elementary numerical methods textbook for the mathematics underlying Newton's method.

Term Structure of Interest Rates

The formulas and methodology for term structure functions come from:

Fabozzi, Frank J. "The Structure of Interest Rates." Ch. 6 in Fabozzi, Frank J. and T. Dessa Fabozzi, eds. *The Handbook of Fixed Income Securities.* 4th ed. New York: Irwin Professional Publishing. 1995. ISBN 0-7863-0001-9.

McEnally, Richard W. and James V. Jordan. "The Term Structure of Interest Rates." Ch. 37 in Fabozzi and Fabozzi, ibid.

Das, Satyajit. "Calculating Zero Coupon Rates." *Swap and Derivative Financing.* Appendix to Ch. 8, pp. 219-225. New York: Irwin Professional Publishing. 1994. ISBN 1-55738-542-4.

Derivatives Pricing and Yields

The pricing and yield formulas for derivative securities come from:

Chriss, Neil A. "Black-Scholes and Beyond: Option Pricing Models," Chicago: Irwin Professional Publishing. 1997. ISBN 0-7863-1025-1.

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

Hull, John C., *Options, Futures, and Other Derivatives*, Prentice Hall, 5th edition, 2003, ISBN 0-13-009056-5

Portfolio Analysis

The Markowitz model is used for portfolio analysis computations. For a discussion of this model see Chapter 7 of:

Bodie, Zvi, Alex Kane, and Alan J. Marcus. *Investments*. Burr Ridge, IL: Irwin. 2nd. ed., 1993, ISBN 0-256-08342-8.

To solve the quadratic minimization problem associated with finding the efficient frontier, the toolbox uses the `fmincon` function (finds the constrained minimum of a function of several variables) in the MATLAB Optimization Toolbox. See that toolbox documentation for more details.

Financial Statistics

The discussion of computing statistical values for portfolios containing missing data elements derives from the following references:

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

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

Sexton, Joe and Anders Rygh Swensen, "ECM Algorithms That Converge at the Rate of EM," *Biometrika*, Vol. 87, No. 3, 2000, pp. 651-662.

Dempster, A. P., N. M. Laird, and Donald B. Rubin, "Maximum Likelihood from Incomplete Data via the EM Algorithm," *Journal of the Royal Statistical Society*, Series B, Vol. 39, No. 1, 1977, pp. 1-37.

Other References

Other references include:

Addendum to Securities Industry Association, *Standard Securities Calculation Methods: Fixed Income Securities Formulas for Analytic Measures,* Vol. 2, Spring 1995. This addendum explains and clarifies the end-of-month rule.

Brealey, Richard A., and Stewart C. Myers. *Principles of Corporate Finance.* New York: McGraw-Hill. 4th ed., 1991, ISBN 0-07-007405-4.

Daigler, Robert T. *Advanced Options Trading.* Chicago: Probus Publishing Co. 1994, ISBN 1-55738-552-1.

A Dictionary of Finance. Oxford: Oxford University Press. 1993, ISBN 0-19-285279-5.

Fabozzi, Frank J., and T. Dessa Fabozzi, eds. *The Handbook of Fixed-Income Securities.* Burr Ridge, IL: Irwin. 4th ed., 1995, ISBN 0-7863-0001-9.

Fitch, Thomas P. *Dictionary of Banking Terms.* Hauppauge, NY: Barron's. 2nd ed., 1993, ISBN 0-8120-1530-4.

Hill, Richard O., Jr. *Elementary Linear Algebra.* Orlando, FL: Academic Press. 1986, ISBN 0-12-348460-X

Luenberger, David G., *Investment Science*, Oxford University Press, 1998. ISBN: 0195108094

Marshall, John F., and Vipul K. Bansal. *Financial Engineering: A Complete Guide to Financial Innovation.* New York: New York Institute of Finance. 1992, ISBN 0-13-312588-2.

Sharpe, William F. *Macro-Investment Analysis*. An "electronic work-in-progress" published on the World Wide Web, 1995, at `http://www.stanford.edu/~wfsharpe/mia/mia.htm`.

Sharpe, William F., and Gordon J. Alexander. *Investments.* Englewood Cliffs, NJ: Prentice-Hall. 4th ed., 1990, ISBN 0-13-504382-4.

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

Glossary

Active return	Amount of return achieved in excess of the return produced by an appropriate benchmark (e.g., an index portfolio).
Active risk	Standard deviation of the active return. Also known as the tracking error.
American option	An option that can be exercised any time until its expiration date. Contrast with European option.
Amortization	Reduction in value of an asset over some period for accounting purposes. Generally used with intangible assets. Depreciation is the term used with fixed or tangible assets.
Annuity	A series of payments over a period of time. The payments are usually in equal amounts and usually at regular intervals such as quarterly, semi-annually, or annually.
Arbitrage	The purchase of securities on one market for immediate resale on another market in order to profit from a price or currency discrepancy.
Basis point	One hundredth of one percentage point, or 0.0001.
Beta	The price volatility of a financial instrument relative to the price volatility of a market or index as a whole. Beta is most commonly used with respect to equities. A high-beta instrument is riskier than a low-beta instrument.
Binomial model	A method of pricing options or other equity derivatives in which the probability over time of each possible price follows a binomial distribution. The basic assumption is that prices can move to only two values (one higher and one lower) over any short time period.
Black-Scholes model	The first complete mathematical model for pricing options, developed by Fischer Black and Myron Scholes. It examines market price, strike price, volatility, time to expiration, and interest rates. It is limited to only certain kinds of options.
Bollinger band chart	A financial chart that plots actual asset data along with three other bands of data: the upper band is two standard deviations above a user-specified moving average; the lower band is two standard deviations below that moving average; and the middle band is the moving average itself.
Bootstrapping, bootstrap method	An arithmetic method for backing an implied zero curve out of the par yield curve.
Building a binomial tree	For a binomial option model: plotting the two possible short-term price-changes values, and then the subsequent two values each, and then the subsequent two values each, and so on over time, is known as "building a binomial tree." See Binomial model.

Call	**a.** An option to buy a certain quantity of a stock or commodity for a specified price within a specified time. See Put. **b.** A demand to submit bonds to the issuer for redemption before the maturity date. **c.** A demand for payment of a debt. **d.** A demand for payment due on stock bought on margin.
Callable bond	A bond that allows the issuer to buy back the bond at a predetermined price at specified future dates. The bond contains an embedded call option; i.e., the holder has sold a call option to the issuer. See Puttable bond.
Candlestick chart	A financial chart usually used to plot the high, low, open, and close price of a security over time. The body of the "candle" is the region between the open and close price of the security. Thin vertical lines extend up to the high and down to the low, respectively. If the open price is greater than the close price, the body is empty. If the close price is greater than the open price, the body is filled. See High-low-close chart.
Cap	Interest-rate option that guarantees that the rate on a floating-rate loan will not exceed a certain level.
Cash flow	Cash received and paid over time.
Collar	Interest-rate option that guarantees that the rate on a floating-rate loan will not exceed a certain upper level nor fall below a lower level. It is designed to protect an investor against wide fluctuations in interest rates.
Convexity	A measure of the rate of change in duration; measured in time. The greater the rate of change, the more the duration changes as yield changes.
Correlation	The simultaneous change in value of two random numeric variables.
Correlation coefficient	A statistic in which the covariance is scaled to a value between minus one (perfect negative correlation) and plus one (perfect positive correlation).
Coupon	Detachable certificate attached to a bond that shows the amount of interest payable at regular intervals, usually semi-annually.Originally coupons were actually attached to the bonds and had to be cut off or "clipped" to redeem them and receive the interest payment.
Coupon dates	The dates when the coupons are paid. Typically a bond pays coupons annually or semi-annually.
Coupon rate	The nominal interest rate that the issuer promises to pay the buyer of a bond.

Covariance	A measure of the degree to which returns on two assets move in tandem. A positive covariance means that asset returns move together; a negative covariance means they vary inversely.
Delta	The rate of change of the price of a derivative security relative to the price of the underlying asset; i.e., the first derivative of the curve that relates the price of the derivative to the price of the underlying security.
Depreciation	Reduction in value of fixed or tangible assets over some period for accounting purposes. See Amortization.
Derivative	A financial instrument that is based on some underlying asset. For example, an option is a derivative instrument based on the right to buy or sell an underlying instrument.
Discount curve	The curve of discount rates vs. maturity dates for bonds.
Duration	The expected life of a fixed-income security considering its coupon yield, interest payments, maturity, and call features. As market interest rates rise, the duration of a financial instrument decreases. See Macaulay duration.
Efficient frontier	A graph representing a set of portfolios that maximizes expected return at each level of portfolio risk. See Markowitz model.
Elasticity	See Lambda.
European option	An option that can be exercised only on its expiration date. Contrast with American option.
Exercise price	The price set for buying an asset (call) or selling an asset (put). The strike price.
Face value	The maturity value of a security. Also known as par value, principal value, or redemption value.
Fixed-income security	A security that pays a specified cash flow over a specific period. Bonds are typical fixed-income securities.
Floor	Interest-rate option that guarantees that the rate on a floating-rate loan will not fall below a certain level.
Forward curve	The curve of forward interest rates vs. maturity dates for bonds.
Forward rate	The future interest rate of a bond inferred from the term structure, especially from the yield curve of zero-coupon bonds, calculated from the growth factor of an investment in a zero held until maturity.

Future value	The value that a sum of money (the present value) earning compound interest will have in the future.
Gamma	The rate of change of delta for a derivative security relative to the price of the underlying asset; i.e., the second derivative of the option price relative to the security price.
Greeks	Collectively, "greeks" refer to the financial measures delta, gamma, lambda, rho, theta, and vega, which are sensitivity measures used in evaluating derivatives.
Hedge	A securities transaction that reduces or offsets the risk on an existing investment position.
High-low-close chart	A financial chart usually used to plot the high, low, open, and close price of a security over time. Plots are vertical lines whose top is the high, bottom is the low, open is a short horizontal tick to the left, and close is a short horizontal tick to the right.
Implied volatility	For an option, the variance that makes a call option price equal to the market price. Given the option price, strike price, and other factors, the Black-Scholes model computes implied volatility.
Internal rate of return	**a.** The average annual yield earned by an investment during the period held. **b.** The effective rate of interest on a loan. **c.** The discount rate in discounted cash flow analysis. **d.** The rate that adjusts the value of future cash receipts earned by an investment so that interest earned equals the original cost. See Yield to maturity.
Issue date	The date a security is first offered for sale. That date usually determines when interest payments, known as coupons, are made.
Ito process	Statistical assumptions about the behavior of security prices. For details, see the book by Hull listed in Appendix A, "Bibliography."
Lambda	The percentage change in the price of an option relative to a 1% change in the price of the underlying security. Also known as Elasticity.
Long position	Outright ownership of a security or financial instrument. The owner expects the price to rise in order to make a profit on some future sale.
Long rate	The yield on a zero-coupon Treasury bond.
Macaulay duration	A widely used measure of price sensitivity to yield changes developed by Frederick Macaulay in 1938. It is measured in years and is a weighted

average-time-to-maturity of an instrument. The Macaulay duration of an income stream, such as a coupon bond, measures how long, on average, the owner waits before receiving a payment. It is the weighted average of the times payments are made, with the weights at time T equal to the present value of the money received at time T.

Markowitz model A model for selecting an optimum investment portfolio, devised by H. M. Markowitz. It uses a discrete-time, continuous-outcome approach for modeling investment problems, often called the mean-variance paradigm. See Efficient frontier.

Maturity date The date when the issuer returns the final face value of a bond to the buyer.

Mean **a.** A number that typifies a set of numbers, such as a geometric mean or an arithmetic mean. **b.** The average value of a set of numbers.

Modified duration The Macaulay duration discounted by the per-period interest rate; i.e., divided by (1+rate/frequency).

Monte-Carlo simulation A mathematical modeling process. For a model that has several parameters with statistical properties, pick a set of random values for the parameters and run a simulation. Then pick another set of values, and run it again. Run it many times (often 10,000 times) and build up a statistical distribution of outcomes of the simulation. This distribution of outcomes is then used to answer whatever question you are asking.

Moving average A price average that is adjusted by adding other parametrically determined prices over some time period.

Moving-averages chart A financial chart that plots leading and lagging moving averages for prices or values of an asset.

Normal (bell-shaped) distribution In statistics, a theoretical frequency distribution for a set of variable data, usually represented by a bell-shaped curve symmetrical about the mean.

Odd first or last period Fixed-income securities may be purchased on dates that do not coincide with coupon or payment dates. The length of the first and last periods may differ from the regular period between coupons, and thus the bond owner is not entitled to the full value of the coupon for that period. Instead, the coupon is pro-rated according to how long the bond is held during that period.

Option	A right to buy or sell specific securities or commodities at a stated price (exercise or strike price) within a specified time. An option is a type of derivative.
Par value	The maturity or face value of a security or other financial instrument.
Par yield curve	The yield curve of bonds selling at par, or face, value.
Point and figure chart	A financial chart usually used to plot asset price data. Upward price movements are plotted as X's and downward price movements are plotted as O's.
Present value	Today's value of an investment that yields some future value when invested to earn compounded interest at a known interest rate.; i.e., the future value at a known period in time discounted by the interest rate over that time period.
Principal value	See Par value.
Purchase price	Price actually paid for a security. Typically the purchase price of a bond is not the same as the redemption value.
Put	An option to sell a stipulated amount of stock or securities within a specified time and at a fixed exercise price. See Call.
Puttable bond	A bond that allows the holder to redeem the bond at a predetermined price at specified future dates. The bond contains an embedded put option; i.e., the holder has bought a put option. See Callable bond.
Quant	A quantitative analyst; someone who does numerical analysis of financial information in order to detect relationships, disparities, or patterns that can lead to making money.
Redemption value	See Par value.
Regression analysis	Statistical analysis techniques that quantify the relationship between two or more variables. The intent is quantitative prediction or forecasting, particularly using a small population to forecast the behavior of a large population.
Rho	The rate of change in a derivative's price relative to the underlying security's risk-free interest rate.
Sensitivity	The "what if" relationship between variables; the degree to which changes in one variable cause changes in another variable. A specific synonym is volatility.

Settlement date The date when money first changes hands; i.e., when a buyer actually pays for a security. It need not coincide with the issue date.

Short rate The annualized one-period interest rate.

Short sale, short position The sale of a security or financial instrument not owned, in anticipation of a price decline and making a profit by purchasing the instrument later at a lower price, and then delivering the instrument to complete the sale. See Long position.

Spot curve, spot yield curve See Zero curve.

Spot rate The current interest rate appropriate for discounting a cash flow of some given maturity.

Spread For options, a combination of call or put options on the same stock with differing exercise prices or maturity dates.

Standard deviation A measure of the variation in a distribution, equal to the square root of the arithmetic mean of the squares of the deviations from the arithmetic mean; the square root of the variance.

Stochastic Involving or containing a random variable or variables; involving chance or probability.

Straddle A strategy used in trading options or futures. It involves simultaneously purchasing put and call options with the same exercise price and expiration date, and it is most profitable when the price of the underlying security is very volatile.

Strike Exercise a put or call option.

Strike price See Exercise price.

Swap A contract between two parties to exchange cash flows in the future according to some formula.

Swaption A swap option; an option on an interest-rate swap. The option gives the holder the right to enter into a contracted interest-rate swap at a specified future date. See Swap.

Term structure The relationship between the yields on fixed-interest securities and their maturity dates. Expectation of changes in interest rates affects term structure, as do liquidity preferences and hedging pressure. A yield curve is one representation in the term structure.

Theta	The rate of change in the price of a derivative security relative to time. Theta is usually very small or negative since the value of an option tends to drop as it approaches maturity.
Tracking error	See active risk.
Treasury bill	Short-term U.S. government security issued at a discount from the face value and paying the face value at maturity.
Treasury bond	Long-term debt obligation of the U.S. government that makes coupon payments semi-annually and is sold at or near par value in $1000 denominations or higher. Face value is paid at maturity.
Variance	The dispersion of a variable. The square of the standard deviation.
Vega	The rate of change in the price of a derivative security relative to the volatility of the underlying security. When vega is large the security is sensitive to small changes in volatility.
Volatility	**a.** Another general term for sensitivity. **b.** The standard deviation of the annualized continuously compounded rate of return of an asset. **c.** A measure of uncertainty or risk.
Yield	**a.** Measure of return on an investment, stated as a percentage of price. Yield can be computed by dividing return by purchase price, current market value, or other measure of value. **b.** Income from a bond expressed as an annualized percentage rate. **c.** The nominal annual interest rate that gives a future value of the purchase price equal to the redemption value of the security. Any coupon payments determine part of that yield.
Yield curve	Graph of yields (vertical axis) of a particular type of security versus the time to maturity (horizontal axis). This curve usually slopes upward, indicating that investors usually expect to receive a premium for securities that have a longer time to maturity. The benchmark yield curve is for U.S. Treasury securities with maturities ranging from three months to 30 years. See Term structure.
Yield to maturity	A measure of the average rate of return that will be earned on a bond if held to maturity.
Zero curve, zero-coupon yield curve	A yield curve for zero-coupon bonds; zero rates versus maturity dates. Since the maturity and duration (Macaulay duration) are identical for zeros, the zero curve is a pure depiction of supply/demand conditions for loanable funds across a continuum of durations and maturities. Also known as spot curve or spot yield curve.

Zero-coupon bond, or Zero	A bond that, instead of carrying a coupon, is sold at a discount from its face value, pays no interest during its life, and pays the principal only at maturity.

Index

Numerics

A

B

C

D

G

H

I

L

R

S