Financial Derivatives 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.	Mail
	3 Apple Hill Drive	
	Natick, MA 01760-2098	

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

Contents

Hedging Portfolios

3

Function Reference

4

Glossary

A

Preface

About This Book

This book describes the Financial Derivatives Toolbox for MATLAB®, a collection of tools for analyzing individual financial derivative instruments and portfolios of instruments.

Organization of the Document

Chapter	Description
"Getting Started"	Describes interest rate models, bushy and recombinent trees, instrument types, and instrument portfolio construction.
"Using Financial Derivatives"	Describes techniques for computing prices and sensitivities based upon the interest rate term structure, the Heath-Jarrow-Morton (HJM) model of forward rates, and the Black-Derman-Toy (BDT) interest rate model.
"Hedging Portfolios"	Describes functions that minimize the cost of hedging a portfolio given a set of target sensitivities, or minimize portfolio sensitivities for a given set of maximum target costs.
"Function Reference"	Describes the functions used for interest rate environment computations, instrument portfolio construction and manipulation, and for Heath-Jarrow-Morton and Black-Derman-Toy modeling.

Typographical Conventions

This manual uses some or all of these conventions.

Item	Convention Used	Example
Example code	Monospace font	To assign the value 5 to A, enter `A = 5`
Function names/syntax	Monospace font	The cos function finds the cosine of each array element. Syntax line example is `MLGetVar ML_var_name`
Keys	**Boldface** with an initial capital letter	Press the **Enter** key.
Literal strings (in syntax descriptions in reference chapters)	**`Monospace bold`** for literals	`f = freqspace(n,'`**`whole`**`')`
Mathematical expressions	*Italics* for variables Standard text font for functions, operators, and constants	This vector represents the polynomial $p = x^2 + 2x + 3$

MATLAB output	Monospace font	MATLAB responds with
		
		`A =`
		`    5`

Menu titles, menu items, dialog boxes, and controls	**Boldface** with an initial capital letter	Choose the **File** menu.
New terms	*Italics*	An *array* is an ordered collection of information.

Item	Convention Used	Example
Omitted input arguments	(...) ellipsis denotes all of the input/output arguments from preceding syntaxes.	`[c,ia,ib] = union(...)`
String variables (from a finite list)	*Monospace italics*	`sysc = d2c(sysd,'`*`method`*`')`

Related Products

The MathWorks provides several products relevant to the tasks you can perform with the Financial Derivatives Toolbox.

For more information about any of these products, see either:

- The online documentation for that product, if it is installed or if you are reading the documentation from the CD
- The MathWorks Web site, at `http://www.mathworks.com`; see the "products" section.

Note The toolboxes listed below all include functions that extend the capabilities of MATLAB.

Product	Description
Database Toolbox	Exchange data with relational databases
Datafeed Toolbox	Acquire real-time financial data from data service providers
Excel Link	Use MATLAB with Microsoft Excel
Financial Time Series Toolbox	Analyze and manage financial time series data
Financial Toolbox	Model financial data and develop financial analysis algorithms
GARCH Toolbox	Analyze financial volatility using univariate GARCH models
MATLAB	The Language of Technical Computing
MATLAB Compiler	Convert MATLAB M-files to C and C++ code

Product	Description
MATLAB Report Generator	Automatically generate documentation for MATLAB applications and data
MATLAB Runtime Server	Deploy runtime versions of MATLAB applications
Optimization Toolbox	Solve standard and large-scale optimization problems
Spline Toolbox	Create and manipulate spline approximation models of data
Statistics Toolbox	Apply statistical algorithms and probability models

Background Reading

Black-Derman-Toy (BDT) Modeling

A description of the Black-Derman-Toy interest rate model can be found in:

Black, Fischer, Emanuel Derman, and William Toy, "A One Factor Model of Interest Rates and its Application to Treasury Bond Options," *Financial Analysts Journal*, January - February 1990.

Heath-Jarrow-Morton (HJM) Modeling

An introduction to Heath-Jarrow-Morton modeling, used extensively in the Financial Derivatives Toolbox, can be found in:

Jarrow, Robert A., *Modelling Fixed Income Securities and Interest Rate Options*, McGraw-Hill, 1996, ISBN 0-07-912253-1.

Financial Derivatives

Information on the creation of financial derivatives and their role in the marketplace can be found in numerous sources. Among those consulted in the development of the Financial Derivatives toolbox are:

Chance, Don. M., *An Introduction to Derivatives*, The Dryden Press, 1998, ISBN 0-030-024483-8

Fabozzi, Frank J., *Treasury Securities and Derivatives*, Frank J. Fabozzi Associates, 1998, ISBN 1-883249-23-6

Hull, John C., *Options, Futures, and Other Derivatives*, Prentice-Hall, 1997, ISBN 0-13-186479-3

Wilmott, Paul, *Derivatives: The Theory and Practice of Financial Engineering*, John Wiley and Sons, 1998, ISBN 0-471-983-89-6

1

Getting Started

Introduction

The Financial Derivatives Toolbox extends the Financial Toolbox in the areas of fixed income derivatives and of securities contingent upon interest rates. The toolbox provides components for analyzing individual financial derivative instruments and portfolios. Specifically, it provides the necessary functions for calculating prices and sensitivities, for hedging, and for visualizing results.

Interest Rate Models

The Financial Derivatives Toolbox computes pricing and sensitivities of interest rate contingent claims based upon:

- The interest rate term structure (sets of zero coupon bonds)
- The Heath-Jarrow-Morton (HJM) model of the interest rate term structure. This model considers a given initial term structure of interest rates and a specification of the volatility of forward rates to build a tree representing the evolution of the interest rates, based upon a statistical process.
- The Black-Derman-Toy (BDT) model for pricing interest rate derivatives. In the BDT model all security prices and rates depend upon the short rate (annualized one-period interest rate). The model uses long rates and their volatilities to construct a tree of possible future short rates. It then determines the value of interest rate sensitive securities from this tree.

For information, see:

- "Pricing and Sensitivity from Interest Rate Term Structure" on page 2-16 for a discussion of price and sensitivity based upon portfolios of zero coupon bonds.
- "Pricing and Sensitivity from HJM" on page 2-34 for a discussion of price and sensitivity based upon the HJM model.
- "Pricing and Sensitivity from BDT" on page 2-63 for a discussion of price and sensitivity based upon the BDT model.

Trees

The Heath-Jarrow-Morton model works with a type of interest rate tree called a *bushy tree*. A bushy tree is a tree in which the number of branches increases exponentially relative to observation times; branches never recombine.

The Black-Derman-Toy model, on the other hand, works with a *recombining tree.* A recombining tree is the opposite of a bushy tree. A recombining tree has branches that recombine over time. From any given node, the node reached by taking the path up-down is the same node reached by taking the path down-up. A bushy and a recombining tree are illustrated below.
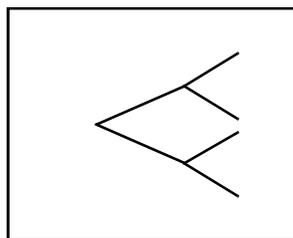
Bushy Tree
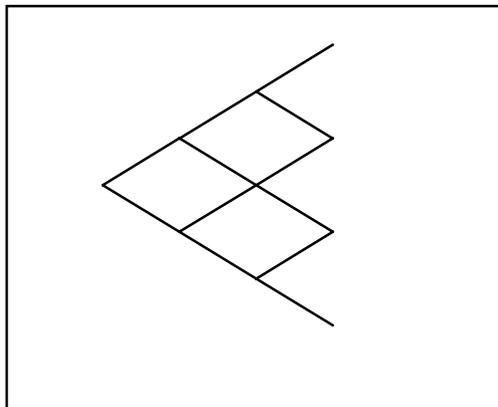
Recombining Tree

This toolbox provides the data file `deriv.mat` that contains two trees, `HJMTree`, a bushy tree, and `BDTTree`, a recombining tree. The toolbox also provides the `treeviewer` function, which graphically displays the shape and data of price, interest rate, and cash flow trees. Viewed with `treeviewer`, the bushy shape of an HJM tree and the recombining shape of a BDT tree are apparent.
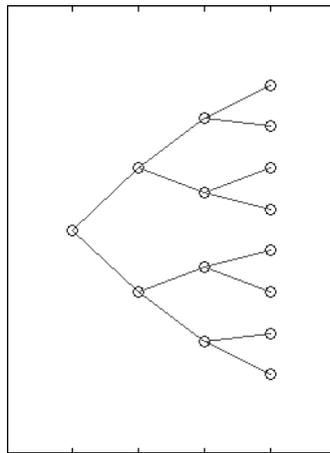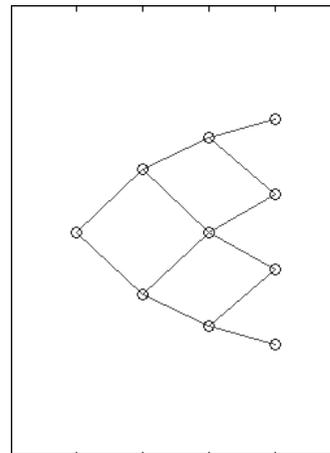
HJMTree (bushy)   **BDTTree (recombining)**

Financial Instruments

The toolbox provides a set of functions that perform computations upon portfolios containing up to seven types of financial instruments.

Bond. A long-term debt security with preset interest rate and maturity, by which the principal and interest must be paid.

Bond Options. Puts and calls on portfolios of bonds.

Fixed Rate Note. A long-term debt security with preset interest rate and maturity, by which the interest must be paid. The principal may or may not be paid at maturity. In this version of the Financial Derivatives Toolbox, the principal is always paid at maturity.

Floating Rate Note. A security similar to a bond, but in which the note's interest rate is reset periodically, relative to a reference index rate, to reflect fluctuations in market interest rates.

Cap. A contract which includes a guarantee that sets the maximum interest rate to be paid by the holder, based upon an otherwise floating interest rate.

Floor. A contract which includes a guarantee setting the minimum interest rate to be received by the holder, based upon an otherwise floating interest rate.

Swap. A contract between two parties obligating the parties to exchange future cash flows. This version of the Financial Derivatives Toolbox handles only the vanilla swap, which is composed of a floating rate leg and a fixed rate leg.

Additionally, the toolbox provides functions for the creation and pricing of *arbitrary cash flow instruments* based upon zero coupon bonds or upon the BDT or HJM models.

Hedging

The Financial Derivatives Toolbox also includes hedging functionality, allowing the rebalancing of portfolios to reach target costs or target sensitivities, which may be set to zero for a neutral-sensitivity portfolio. Optionally, the rebalancing process can be self-financing or directed by a set of user-supplied constraints. For information, see:

- "Hedging" on page 3-2 for a discussion of the hedging process.
- "hedgeopt" on page 4-74 for a description of the function that allocates an optimal hedge.
- "hedgeslf" on page 4-77 for a description of the function that allocates a self-financing hedge.

Creating and Managing Instrument Portfolios

The Financial Derivatives Toolbox provides components for analyzing individual derivative instruments and portfolios containing several types of financial instruments. The toolbox provides functionality that supports the creation and management of these instruments:

- Bonds
- Bond Options
- Fixed Rate Notes
- Floating Rate Notes
- Caps
- Floors
- Swaps

Additionally, the toolbox provides functions for the creation of *arbitrary cash flow instruments*.

The toolbox also provides pricing and sensitivity routines for these instruments. (See "Pricing and Sensitivity from Interest Rate Term Structure" on page 2-16, "Pricing and Sensitivity from HJM" on page 2-34, or "Pricing and Sensitivity from BDT" on page 2-63 for information.)

Portfolio Creation

The `instadd` function creates a set of instruments (portfolio) or adds instruments to an existing instrument collection. The `TypeString` argument specifies the type of the investment instrument: `'Bond'`, `'OptBond'`, `'CashFlow'`, `'Fixed'`, `'Float'`, `'Cap'`, `'Floor'`, or `'Swap'`. The input arguments following `TypeString` are specific to the type of investment instrument. Thus, the `TypeString` argument determines how the remainder of the input arguments is interpreted.

For example, `instadd` with the type string `'Bond'` creates a portfolio of bond instruments

```
InstSet = instadd('Bond', CouponRate, Settle, Maturity, Period,
Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate,
StartDate, Face)
```

In a similar manner, `instadd` can create portfolios of other types of investment instruments:

- Bond option

  ```
  InstSet = instadd('OptBond', BondIndex, OptSpec, Strike,
  ExerciseDates, AmericanOpt)
  ```

- Arbitrary cash flow instrument

  ```
  InstSet = instadd('CashFlow', CFlowAmounts, CFlowDates, Settle,
  Basis)
  ```

- Fixed rate note instrument

  ```
  InstSet = instadd('Fixed', CouponRate, Settle, Maturity,
  FixedReset, Basis, Principal)
  ```

- Floating rate note instrument

  ```
  InstSet = instadd('Float', Spread, Settle, Maturity, FloatReset,
  Basis, Principal)
  ```

- Cap instrument

  ```
  InstSet = instadd('Cap', Strike, Settle, Maturity, CapReset,
  Basis, Principal)
  ```

- Floor instrument

  ```
  InstSet = instadd('Floor', Strike, Settle, Maturity, FloorReset,
  Basis, Principal)
  ```

- Swap instrument

  ```
  InstSet = instadd('Swap', LegRate, Settle, Maturity, LegReset,
  Basis, Principal, LegType)
  ```

To use the `instadd` function to add additional instruments to an existing instrument portfolio, provide the name of an existing portfolio as the first argument to the `instadd` function.

Consider, for example, a portfolio containing two cap instruments only.

```
Strike = [0.06; 0.07];
Settle = '08-Feb-2000';
Maturity = '15-Jan-2003';
```

```
Port_1 = instadd('Cap', Strike, Settle, Maturity);
```

These commands create a portfolio containing two cap instruments with the same settlement and maturity dates, but with different strikes. In general, the input arguments describing an instrument can be either a scalar, or a number of instruments (NumInst)-by-1 vector in which each element corresponds to an instrument. Using a scalar assigns the same value to all instruments passed in the call to instadd.

Use the instdisp command to display the contents of the instrument set.

```
instdisp(Port_1)

Index Type Strike Settle      Maturity    CapReset Basis
Principal
1    Cap  0.06   08-Feb-2000 15-Jan-2003 NaN      NaN   NaN
2    Cap  0.07   08-Feb-2000 15-Jan-2003 NaN      NaN   NaN
```

Now add a single bond instrument to Port_1. The bond has a 4.0% coupon and the same settlement and maturity dates as the cap instruments.

```
CouponRate = 0.04;
Port_1 = instadd(Port_1, 'Bond', CouponRate, Settle, Maturity);
```

Use instdisp again to see the resulting instrument set.

```
instdisp(Port_1)

Index Type Strike Settle      Maturity    CapReset Basis
Principal
1    Cap  0.06   08-Feb-2000 15-Jan-2003 NaN      NaN   NaN
2    Cap  0.07   08-Feb-2000 15-Jan-2003 NaN      NaN   NaN

Index Type CouponRate Settle      Maturity    Period Basis ...
3    Bond 0.04       08-Feb-2000 15-Jan-2003 NaN    NaN ...
```

Portfolio Management

The portfolio management capabilities provided by the Financial Derivatives toolbox include:

- Constructors for the most common financial instruments. (See "Instrument Constructors" on page 1-9.)
- The ability to create new instruments or to add new fields to existing instruments. (See "Creating New Instruments or Properties" on page 1-10.)
- The ability to search or subset a portfolio. See "Searching or Subsetting a Portfolio" on page 1-12.)

Instrument Constructors

The toolbox provides constructors for the most common financial instruments.

Note A *constructor* is a function that builds a structure dedicated to a certain type of object; in this toolbox, an *object* is a type of market instrument.

The instruments and their constructors in this toolbox are listed below.

Instrument	Constructor
Bond	`instbond`
Bond option	`instoptbnd`
Arbitrary cash flow	`instcf`
Fixed rate note	`instfixed`
Floating rate note	`instfloat`
Cap	`instcap`
Floor	`instfloor`
Swap	`instswap`

Each instrument has parameters (fields) that describe the instrument. The toolbox functions enable you to:

- Create an instrument or portfolio of instruments
- Enumerate stored instrument types and information fields
- Enumerate instrument field data
- Search and select instruments

The instrument structure consists of various fields according to instrument type. A *field* is an element of data associated with the instrument. For example, a bond instrument contains the fields CouponRate, Settle, Maturity, etc. Additionally, each instrument has a field that identifies the investment type (bond, cap, floor, etc.).

In reality the set of parameters for each instrument is not fixed. Users have the ability to add additional parameters. These additional fields will be ignored by the toolbox functions. They may be used to attach additional information to each instrument, such as an internal code describing the bond.

Parameters not specified when *creating* an instrument default to NaN, which, in general, means that the functions using the instrument set (such as intenvprice or hjmprice) will use default values. At the time of *pricing*, an error occurs if any of the required fields is missing, such as Strike in a cap, or the CouponRate in a bond.

Creating New Instruments or Properties

Use the instaddfield function to create a new kind of instrument or to add new properties to the instruments in an existing instrument collection.

To create a new kind of instrument with instaddfield, you need to specify three arguments: 'Type', 'FieldName', and 'Data'. 'Type' defines the type of the new instrument, for example, Future. 'FieldName' names the fields uniquely associated with the new type of instrument. 'Data' contains the data for the fields of the new instrument.

An optional fourth parameter is 'ClassList'. 'ClassList' specifies the data types of the contents of each unique field for the new instrument.

Here are the syntaxes to create a new kind of instrument using instaddfield.

```
InstSet = instaddfield('FieldName', FieldList, 'Data', DataList,
'Type', TypeString)

InstSet = instaddfield('FieldName', FieldList, 'FieldClass',
ClassList, 'Data' , DataList, 'Type', TypeString)
```

To add new instruments to an existing set, use

```
InstSetNew = instaddfield(InstSetOld, 'FieldName', FieldList,
'Data', DataList, 'Type', TypeString)
```

As an example, consider a futures contract with a delivery date of July 15, 2000, and a quoted price of $104.40. Since the Financial Derivatives Toolbox does not directly support this instrument, you must create it using the function `instaddfield`. The parameters used for the creation of the instruments are:

• Type: `Future`
• Field names: `Delivery` and `Price`
• Data: Delivery is July 15, 2000, and Price is $104.40.

Enter the data into MATLAB.

```
Type = 'Future';
FieldName = {'Delivery', 'Price'};
Data = {'Jul-15-2000', 104.4};
```

Optionally, you can also specify the data types of the data cell array by creating another cell array containing this information.

```
FieldClass = {'date','dble'};
```

Finally, create the portfolio with a single instrument.

```
Port = instaddfield('Type', Type, 'FieldName', FieldName,...
'FieldClass', FieldClass, 'Data', Data);
```

Now use the function `instdisp` to examine the resulting single-instrument portfolio.

```
instdisp(Port)

Index   Type   Delivery      Price
1       Future 15-Jul-2000   104.4
```

Because your portfolio `Port` has the same structure as those created using the function `instadd`, you can combine portfolios created using `instadd` with portfolios created using `instaddfield`. For example, you can now add two cap instruments to `Port` with `instadd`.

```
Strike = [0.06; 0.07];
Settle = '08-Feb-2000';
Maturity = '15-Jan-2003';

Port = instadd(Port, 'Cap', Strike, Settle, Maturity);
```

View the resulting portfolio using `instdisp`.

```
instdisp(Port)

Index   Type   Delivery       Price
1          Future 15-Jul-2000   104.4

Index Type Strike Settle       Maturity    CapReset  Basis Pricip
al
2      Cap  0.06   08-Feb-2000 15-Jan-2003 NaN        NaN   NaN
3      Cap  0.07   08-Feb-2000 15-Jan-2003 NaN        NaN   NaN
```

Searching or Subsetting a Portfolio

The Financial Derivatives Toolbox provides functions that enable you to:

- Find specific instruments within a portfolio
- Create a subset portfolio consisting of instruments selected from a larger portfolio

The `instfind` function finds instruments with a specific parameter value; it returns an instrument index (position) in a large instrument set. The `instselect` function, on the other hand, subsets a large instrument set into a portfolio of instruments with designated parameter values; it returns an instrument set (portfolio) rather than an index.

instfind. The general syntax for `instfind` is

```
IndexMatch = instfind(InstSet, 'FieldName', FieldList, 'Data',
DataList, 'Index', IndexSet, 'Type', TypeList)
```

`InstSet` is the instrument set to search. Within `InstSet` instruments are categorized by type, and each type can have different data fields. The stored data field is a row vector or string for each instrument.

The `FieldList`, `DataList`, and `TypeList` arguments indicate values to search for in the `'FieldName'`, `'Data'`, and `'Type'` data fields of the instrument set. `FieldList` is a cell array of field name(s) specific to the instruments. `DataList` is a cell array or matrix of acceptable values for the parameter(s) specified in `FieldList`. `'FieldName'` and `'Data'` (consequently, `FieldList` and `DataList`) parameters must appear together or not at all.

IndexSet is a vector of integer index(es) designating positions of instruments in the instrument set to check for matches; the default is all indices available in the instrument set. 'TypeList' is a string or cell array of strings restricting instruments to match one of the 'TypeList' types; the default is all types in the instrument set.

IndexMatch is a vector of positions of instruments matching the input criteria. Instruments are returned in IndexMatch if all the 'FieldName', 'Data', 'Index', and 'Type' conditions are met. An instrument meets an individual field condition if the stored 'FieldName' data matches any of the rows listed in the DataList for that FieldName.

instfind Examples. The examples use the provided MAT-file deriv.mat.

The MAT-file contains an instrument set, HJMInstSet, that contains eight instruments of seven types.

```
load deriv.mat
instdisp(HJMInstSet)
```

```
Index Type CouponRate Settle      Maturity    Period Basis .........
          Name       Quantity
1    Bond 0.04        01-Jan-2000 01-Jan-2003  1
     NaN.........                 4% bond   100
2    Bond 0.04        01-Jan-2000 01-Jan-2004  2
     NaN.........                 4% bond    50


Index Type    UnderInd OptSpec Strike ExerciseDates  AmericanOpt Name
        Quantity
3    OptBond 2        call    101    01-Jan-2003    NaN         Option 101  -50

Index Type  CouponRate Settle     Maturity      FixedReset Basis Principal Name
     Quantity
4    Fixed 0.04        01-Jan-2000 01-Jan-2003   1          NaN   NaN        4%
Fixed 80

Index Type  Spread Settle     Maturity    FloatReset  Basis Principal Name
        Quantity
5    Float 20      01-Jan-2000 01-Jan-2003 1          NaN   NaN       20BP Float 8

Index Type Strike Settle        Maturity      CapReset Basis Principal Name
Quantity
6    Cap  0.03   01-Jan-2000    01-Jan-2004    1        NaN   NaN       3% Cap 30

Index Type  Strike Settle       Maturity      FloorReset Basis Principal Name
    Quantity
7    Floor 0.03 01-Jan-2000 01-Jan-2004    1            NaN   NaN       3% Floor 40

Index Type LegRate    Settle       Maturity     LegReset Basis Principal LegType
Name          Quantity
8    Swap [0.06 20] 01-Jan-2000   01-Jan-2003   [1  1]   NaN   NaN        [NaN]
  6%/20BP Swap 10
```

Find all instruments with a maturity date of January 01, 2003.

```
Mat2003 = ...
instfind(HJMInstSet,'FieldName','Maturity','Data','01-Jan-2003')
```

```
Mat2003 =

     1
     4
     5
     8
```

Find all cap and floor instruments with a maturity date of January 01, 2004.

```
CapFloor = instfind(HJMInstSet,...
'FieldName','Maturity','Data','01-Jan-2004', 'Type',...
{'Cap';'Floor'})

CapFloor =

     6
     7
```

Find all instruments where the portfolio is long or short a quantity of 50.

```
Pos50 = instfind(HJMInstSet,'FieldName',...
'Quantity','Data',{'50';'-50'})

Pos50 =

     2
     3
```

instselect. The syntax for `instselect` is exactly the same syntax as for `instfind`. `instselect` returns a full portfolio instead of indexes into the original portfolio. Compare the values returned by both functions by calling them equivalently.

Previously you used `instfind` to find all instruments in `HJMInstSet` with a maturity date of January 01, 2003.

```
Mat2003 = ...
instfind(HJMInstSet,'FieldName','Maturity','Data','01-Jan-2003')

Mat2003 =

     1
```

```
                        4
                        5
                        8
```

Now use the same instrument set as a starting point, but execute the `instselect` function instead, to produce a new instrument set matching the identical search criteria.

```
Select2003 = ...
instselect(HJMInstSet,'FieldName','Maturity','Data',...
'01-Jan-2003')

instdisp(Select2003)
```

```
Index Type CouponRate Settle        Maturity      Period Basis .........
          Name       Quantity
1     Bond 0.04        01-Jan-2000 01-Jan-2003  1
      NaN.........                   4% bond   100

Index Type  CouponRate Settle       Maturity       FixedReset Basis Principal Name
     Quantity
2     Fixed 0.04         01-Jan-2000 01-Jan-2003    1          NaN   NaN        4%
Fixed 80

Index Type  Spread Settle       Maturity     FloatReset  Basis Principal Name
      Quantity
3     Float 20      01-Jan-2000 01-Jan-2003 1          NaN   NaN       20BP Float 8

Index Type LegRate    Settle       Maturity     LegReset Basis Principal LegType
Name         Quantity
4     Swap [0.04 20] 01-Jan-2000   01-Jan-2003 [1  1]   NaN   NaN       [NaN]
  4%/20BP Swap 10
```

instselect Examples. These examples use the portfolio `ExampleInst` provided with the MAT-file `InstSetExamples.mat`.

```
load InstSetExamples.mat
instdisp(ExampleInst)

Index Type    Strike Price Opt  Contracts
1      Option 95     12.2 Call     0
2      Option 100     9.2 Call     0
3      Option 105     6.8 Call  1000

Index Type     Delivery       F     Contracts
4      Futures 01-Jul-1999    104.4 -1000

Index Type    Strike Price Opt  Contracts
5      Option 105     7.4 Put  -1000
6      Option 95      2.9 Put      0

Index Type  Price Maturity      Contracts
7      TBill 99    01-Jul-1999   6
```

The instrument set contains three instrument types: `'Option'`, `'Futures'`, and `'TBill'`. Use `instselect` to make a new instrument set containing only options struck at 95. In other words, select all instruments containing the field `Strike` *and* with the data value for that field equal to 95.

```
InstSet = instselect(ExampleInst,'FieldName','Strike','Data',95)

instdisp(InstSet)

Index Type    Strike Price Opt  Contracts
1      Option 95     12.2 Call     0
2      Option 95      2.9 Put      0
```

You can use all the various forms of `instselect` and `instfind` to locate specific instruments within this instrument set.

2

Using Financial Derivatives

Interest Rate Environment

The *interest rate term structure* is the representation of the evolution of interest rates through time. In MATLAB, the interest rate environment is encapsulated in a structure called RateSpec (*rate specification*). This structure holds all information needed to identify completely the evolution of interest rates. Several functions included in the Financial Derivatives Toolbox are dedicated to the creation and management of the RateSpec structure. Many others take this structure as an input argument representing the evolution of interest rates.

Before looking further at the RateSpec structure, examine three functions that provide key functionality for working with interest rates: disc2rate, its opposite, rate2disc, and ratetimes. The first two functions map between discount rates and interest rates. The third function, ratetimes, calculates the effect of term changes on the interest rates.

Interest Rates vs. Discount Factors

Discount factors are coefficients commonly used to find the present value of future cash flows. As such, there is a direct mapping between the rate applicable to a period of time, and the corresponding discount factor. The function disc2rate converts discount rates for a given term (period) into interest rates. The function rate2disc does the opposite; it converts interest rates applicable to a given term (period) into the corresponding discount rates.

Calculating Discount Factors from Rates

As an example, consider these annualized zero coupon bond rates.

From	To	Rate
15 Feb 2000	15 Aug 2000	0.05
15 Feb 2000	15 Feb 2001	0.056
15 Feb 2000	15 Aug 2001	0.06
15 Feb 2000	15 Feb 2002	0.065
15 Feb 2000	15 Aug 2002	0.075

To calculate the discount factors corresponding to these interest rates, call `rate2disc` using the syntax

```
Disc = rate2disc(Compounding, Rates, EndDates, StartDates,
ValuationDate)
```

where:

- `Compounding` represents the frequency at which the zero rates are compounded when annualized. For this example, assume this value to be 2.
- `Rates` is a vector of annualized percentage rates representing the interest rate applicable to each time interval.
- `EndDates` is a vector of dates representing the end of each interest rate term (period).
- `StartDates` is a vector of dates representing the beginning of each interest rate term.
- `ValuationDate` is the date of observation for which the discount factors are calculated. In this particular example, use February 15, 2000 as the beginning date for all interest rate terms.

Set the variables in MATLAB.

```
StartDates = ['15-Feb-2000'];
EndDates  = ['15-Aug-2000'; '15-Feb-2001'; '15-Aug-2001';...
'15-Feb-2002'; '15-Aug-2002'];
Compounding = 2;
ValuationDate = ['15-Feb-2000'];
Rates = [0.05; 0.056; 0.06; 0.065; 0.075];
Disc = rate2disc(Compounding, Rates, EndDates, StartDates,...
ValuationDate)

Disc =

    0.9756
    0.9463
    0.9151
    0.8799
    0.8319
```

By adding a fourth column to the above rates table to include the corresponding discounts, you can see the evolution of the discount rates.

From	To	Rate	Discount
15 Feb 2000	15 Aug 2000	0.05	0.9756
15 Feb 2000	15 Feb 2001	0.056	0.9463
15 Feb 2000	15 Aug 2001	0.06	0.9151
15 Feb 2000	15 Feb 2002	0.065	0.8799
15 Feb 2000	15 Aug 2002	0.075	0.8319

Optional Time Factor Outputs

The function rate2disc optionally returns two additional output arguments: EndTimes and StartTimes. These vectors of time factors represent the start dates and end dates in discount periodic units. The scale of these units is determined by the value of the input variable Compounding.

To examine the time factor outputs, find the corresponding values in the previous example.

```
[Disc, EndTimes, StartTimes] = rate2disc(Compounding, Rates,...
EndDates, StartDates, ValuationDate);
```

Arrange the two vectors into a single array for easier visualization.

```
Times = [StartTimes, EndTimes]

Times =

     0     1
     0     2
     0     3
     0     4
     0     5
```

Because the valuation date is equal to the start date for all periods, the StartTimes vector is composed of zeros. Also, since the value of Compounding is 2, the rates are compounded semiannually, which sets the units of periodic discount to six months. The vector EndDates is composed of dates separated by

intervals of six months from the valuation date. This explains why the
EndTimes vector is a progression of integers from one to five.

Alternative Syntax (rate2disc)

The function rate2disc also accommodates an alternative syntax that uses
periodic discount units instead of dates. Since the relationship between
discount factors and interest rates is based on time periods and not on absolute
dates, this form of rate2disc allows you to work directly with time periods. In
this mode, the valuation date corresponds to zero, and the vectors StartTimes
and EndTimes are used as input arguments instead of their date equivalents,
StartDates and EndDates. This syntax for rate2disc is

```
Disc = rate2disc(Compounding, Rates, EndTimes, StartTimes)
```

Using as input the StartTimes and EndTimes vectors computed previously, you
should obtain the previous results for the discount factors.

```
Disc = rate2disc(Compounding, Rates, EndTimes, StartTimes)

Disc =

    0.9756
    0.9463
    0.9151
    0.8799
    0.8319
```

Calculating Rates from Discounts

The function disc2rate is the complement to rate2disc. It finds the rates
applicable to a set of compounding periods, given the discount factor in those
periods. The syntax for calling this function is

```
Rates = disc2rate(Compounding, Disc, EndDates, StartDates,
ValuationDate)
```

Each argument to this function has the same meaning as in rate2disc. Use the
results found in the previous example to return the rate values you started
with.

```
Rates = disc2rate(Compounding, Disc, EndDates, StartDates,...
ValuationDate)
```

```
Rates =

    0.0500
    0.0560
    0.0600
    0.0650
    0.0750
```

Alternative Syntax (disc2rate)

As in the case of `rate2disc`, `disc2rate` optionally returns `StartTimes` and `EndTimes` vectors representing the start and end times measured in discount periodic units. Again, working with the same values as before, you should obtain the same numbers.

```
[Rates, EndTimes, StartTimes] = disc2rate(Compounding, Disc,...
EndDates, StartDates, ValuationDate);
```

Arrange the results in a matrix convenient to display.

```
Result = [StartTimes, EndTimes, Rates]

Result =

        0    1.0000    0.0500
        0    2.0000    0.0560
        0    3.0000    0.0600
        0    4.0000    0.0650
        0    5.0000    0.0750
```

As with `rate2disc`, the relationship between rates and discount factors is determined by time periods and not by absolute dates. Consequently, the alternate syntax for `disc2rate` uses time vectors instead of dates, and it assumes that the valuation date corresponds to time = 0. The times-based calling syntax is

```
Rates = disc2rate(Compounding, Disc, EndTimes, StartTimes);
```

Using this syntax, we again obtain the original values for the interest rates.

```
Rates = disc2rate(Compounding, Disc, EndTimes, StartTimes)

Rates =
```

```
                        0.0500
                        0.0560
                        0.0600
                        0.0650
                        0.0750
```

Interest Rate Term Conversions

Interest rate evolution is typically represented by a set of interest rates, including the beginning and end of the periods the rates apply to. For zero rates, the start dates are typically at the valuation date, with the rates extending from that valuation date until their respective maturity dates.

Calculating Rates Applicable to Different Periods

Frequently, given a set of rates including their start and end dates, you may be interested in finding the rates applicable to different terms (periods). This problem is addressed by the function `ratetimes`. This function interpolates the interest rates given a change in the original terms. The syntax for calling `ratetimes` is

```
  [Rates, EndTimes, StartTimes] = ratetimes(Compounding, RefRates,
  RefEndDates, RefStartDates, EndDates, StartDates, ValuationDate);
```

where:

- `Compounding` represents the frequency at which the zero rates are compounded when annualized.
- `RefRates` is a vector of initial interest rates representing the interest rates applicable to the initial time intervals.
- `RefEndDates` is a vector of dates representing the end of the interest rate terms (period) applicable to `RefRates`.
- `RefStartDates` is a vector of dates representing the beginning of the interest rate terms applicable to `RefRates`.
- `EndDates` represent the maturity dates for which the interest rates are interpolated.
- `StartDates` represent the starting dates for which the interest rates are interpolated.

- `ValuationDate` is the date of observation, from which the `StartTimes` and `EndTimes` are calculated. This date represents time = 0.

The input arguments to this function can be separated into two groups:

- The initial or reference interest rates, including the terms for which they are valid
- Terms for which the new interest rates are calculated

As an example, consider the rate table specified earlier.

From	To	Rate
15 Feb 2000	15 Aug 2000	0.05
15 Feb 2000	15 Feb 2001	0.056
15 Feb 2000	15 Aug 2001	0.06
15 Feb 2000	15 Feb 2002	0.065
15 Feb 2000	15 Aug 2002	0.075

Assuming that the valuation date is February 15, 2000, these rates represent zero coupon bond rates with maturities specified in the second column. Use the function `ratetimes` to calculate the spot rates at the beginning of all periods implied in the table. Assume a compounding value of 2.

```
% Reference Rates.
RefStartDates = ['15-Feb-2000'];
RefEndDates  = ['15-Aug-2000'; '15-Feb-2001'; '15-Aug-2001';...
'15-Feb-2002'; '15-Aug-2002'];
Compounding = 2;
ValuationDate = ['15-Feb-2000'];
RefRates = [0.05; 0.056; 0.06; 0.065; 0.075];

% New Terms.
StartDates = ['15-Feb-2000'; '15-Aug-2000'; '15-Feb-2001';...
'15-Aug-2001'; '15-Feb-2002'];
EndDates  =  ['15-Aug-2000'; '15-Feb-2001'; '15-Aug-2001';...
'15-Feb-2002'; '15-Aug-2002'];
% Find the new rates.
```

```
[Rates, EndTimes, StartTimes] = ratetimes(Compounding, ...
RefRates, RefEndDates, RefStartDates, EndDates, StartDates,...
ValuationDate);

Rates =

    0.0500
    0.0620
    0.0680
    0.0801
    0.1155
```

Place these values in a table similar to the one above. Observe the evolution of the spot rates based on the initial zero coupon rates.

From	To	Rate
15 Feb 2000	15 Aug 2000	0.0500
15 Aug 2000	15 Feb 2001	0.0620
15 Feb 2001	15 Aug 2001	0.0680
15 Aug 2001	15 Feb 2002	0.0801
15 Feb 2002	15 Aug 2002	0.1155

Alternative Syntax (ratetimes)

The additional output arguments StartTimes and EndTimes represent the time factor equivalents to the StartDates and EndDates vectors. As with the functions disc2rate and rate2disc, ratetimes uses time factors for interpolating the rates. These time factors are calculated from the start and end dates, and the valuation date, which are passed as input arguments. ratetimes also has an alternate syntax that uses time factors directly, and assumes time = 0 as the valuation date. This alternate syntax is

```
[Rates, EndTimes, StartTimes] = ratetimes(Compounding, RefRates,
RefEndTimes, RefStartTimes, EndTimes, StartTimes);
```

Use this alternate version of ratetimes to find the spot rates again. In this case, you must first find the time factors of the reference curve. Use date2time for this.

```
RefEndTimes = date2time(ValuationDate, RefEndDates, Compounding)
```

```
RefEndTimes =

    1
    2
    3
    4
    5

RefStartTimes = date2time(ValuationDate, RefStartDates,...
Compounding)

RefStartTimes =

    0
```

These are the expected values, given semiannual discounts (as denoted by a value of 2 in the variable Compounding), end dates separated by six-month periods, and the valuation date equal to the date marking beginning of the first period (time factor = 0).

Now call ratetimes with the alternate syntax.

```
[Rates, EndTimes, StartTimes] = ratetimes(Compounding,...
RefRates, RefEndTimes, RefStartTimes, EndTimes, StartTimes);

Rates =

    0.0500
    0.0620
    0.0680
    0.0801
    0.1155
```

EndTimes and StartTimes have, as expected, the same values they had as input arguments.

```
Times = [StartTimes, EndTimes]

Times =

    0     1
```

```
       1      2
       2      3
       3      4
       4      5
```

Interest Rate Term Structure

The Financial Derivatives Toolbox includes a set of functions to encapsulate interest rate term information into a single structure. These functions present a convenient way to package all information related to interest rate terms into a common format, and to resolve interdependencies when one or more of the parameters is modified. For information, see:

- "Creation or Modification (intenvset)" on page 2-11 for a discussion of how to create or modify an interest rate term structure (RateSpec) using the intenvset function.
- "Obtaining Specific Properties (intenvget)" on page 2-13 for a discussion of how to extract specific properties from a RateSpec.

Creation or Modification (intenvset)

The main function to create or modify an interest rate term structure RateSpec (*rates specification*) is intenvset. If the first argument to this function is a previously created RateSpec, the function modifies the existing rate specification and returns a new one. Otherwise, it creates a new RateSpec. The other intenvset arguments are property-value pairs, indicating the new value for these properties. The properties that can be specified or modified are:

- Compounding
- Disc
- Rates
- EndDates
- StartDates
- ValuationDate
- Basis
- EndMonthRule

To learn about the properties EndMonthRule and Basis, type
help ftbEndMonthRule and help ftbBasis or see the *Financial Toolbox User's Guide*.

Consider again the original table of interest rates.

From	To	Rate
15 Feb 2000	15 Aug 2000	0.05
15 Feb 2000	15 Feb 2001	0.056
15 Feb 2000	15 Aug 2001	0.06
15 Feb 2000	15 Feb 2002	0.065
15 Feb 2000	15 Aug 2002	0.075

Use the information in this table to populate the RateSpec structure.

```
StartDates = ['15-Feb-2000'];
EndDates =   ['15-Aug-2000';
             '15-Feb-2001';
             '15-Aug-2001';
             '15-Feb-2002';
             '15-Aug-2002'];
Compounding = 2;
ValuationDate = ['15-Feb-2000'];
Rates = [0.05; 0.056; 0.06; 0.065; 0.075];

rs = intenvset('Compounding',Compounding,'StartDates',...
StartDates, 'EndDates', EndDates, 'Rates', Rates,...
'ValuationDate', ValuationDate)

rs =

       FinObj:'RateSpec'
  Compounding:2
         Disc:[5x1 double]
        Rates:[5x1 double]
     EndTimes:[5x1 double]
   StartTimes:[5x1 double]
     EndDates:[5x1 double]
```

```
    StartDates:730531
ValuationDate:730531
        Basis: 0
  EndMonthRule: 1
```

Some of the properties filled in the structure were not passed explicitly in the call to RateSpec. The values of the automatically completed properties depend upon the properties that are explicitly passed. Consider for example the StartTimes and EndTimes vectors. Since the StartDates and EndDates vectors are passed in, as well as the ValuationDate, intenvset has all the information needed to calculate StartTimes and EndTimes. Hence, these two properties are read only.

Obtaining Specific Properties (intenvget)

The complementary function to intenvset is intenvget. This function obtains specific properties from the interest rate term structure. The syntax of this function is

```
ParameterValue = intenvget(RateSpec, 'ParameterName')
```

To obtain the vector EndTimes from the RateSpec structure, enter

```
EndTimes = intenvget(rs, 'EndTimes')

EndTimes =

     1
     2
     3
     4
     5
```

To obtain Disc, the values for the discount factors that were calculated automatically by intenvset, type

```
Disc = intenvget(rs, 'Disc')

Disc =

    0.9756
    0.9463
    0.9151
```

```
0.8799
0.8319
```

These discount factors correspond to the periods starting from `StartDates` and ending in `EndDates`.

Note Although you can directly access these fields within the structure instead of using `intenvget`, we strongly advise against this. The format of the interest rate term structure could change in future versions of the toolbox. Should that happen, any code accessing the `RateSpec` fields directly would stop working.

Now use the `RateSpec` structure with its functions to examine how changes in specific properties of the interest rate term structure affect those depending upon it. As an exercise, change the value of `Compounding` from 2 (semiannual) to 1 (annual).

```
rs = intenvset(rs, 'Compounding', 1);
```

Since `StartTimes` and `EndTimes` are measured in units of periodic discount, a change in `Compounding` from 2 to 1 redefines the basic unit from semiannual to annual. This means that a period of six months is represented with a value of 0.5, and a period of one year is represented by 1. To obtain the vectors `StartTimes` and `EndTimes`, enter

```
StartTimes = intenvget(rs, 'StartTimes');
EndTimes = intenvget(rs, 'EndTimes');
Times = [StartTimes, EndTimes]

Times =

        0    0.5000
        0    1.0000
        0    1.5000
        0    2.0000
        0    2.5000
```

Since all the values in `StartDates` are the same as the valuation date, all `StartTimes` values are zero. On the other hand, the values in the `EndDates` vector are dates separated by six-month periods. Since the redefined value of

compounding is 1, `EndTimes` becomes a sequence of numbers separated by increments of 0.5.

Pricing and Sensitivity from Interest Rate Term Structure

The Financial Derivatives Toolbox contains a family of functions that finds the price and sensitivities of several financial instruments based on interest rate curves. For information, see:

- "Pricing" on page 2-17 for a discussion on using the `intenvprice` function to price a portfolio of instruments based on a set of zero curves.
- "Sensitivity" on page 2-19 for a discussion on computing delta and gamma sensitivities with the `intenvsens` function.

The instruments can be presented to the functions as a portfolio of different types of instruments or as groups of instruments of the same type. The current version of the toolbox can compute price and sensitivities for four instrument types using interest rate curves:

- Bonds
- Fixed Rate Notes
- Floating Rate Notes
- Swaps

In addition to these instruments, the toolbox also supports the calculation of price and sensitivities of arbitrary sets of cash flows.

Note that options and interest rates floors and caps are absent from the above list of supported instruments. These instruments are not supported because their pricing and sensitivity function require a stochastic model for the evolution of interest rates. The interest rate term structure used for pricing is treated as deterministic, and as such is not adequate for pricing these instruments.

The Financial Derivatives Toolbox additionally contains functions that use the Heath-Jarrow-Morton (HJM) and Black-Derman-Toy (BDT) models to compute prices and sensitivities for financial instruments. These models support computations involving options and interest rate floors and caps. See "Pricing and Sensitivity from HJM" on page 2-34 and "Pricing and Sensitivity from BDT" on page 2-63 for information on computing price and sensitivities of financial instruments using HJM and BDT models.

Pricing

The main function used for pricing portfolios of instruments is `intenvprice`. This function works with the family of functions that calculate the prices of individual types of instruments. When called, `intenvprice` classifies the portfolio contained in `InstSet` by instrument type, and calls the appropriate pricing functions. The map between instrument types and the pricing function `intenvprice` calls is

`bondbyzero`:	Price bond by a set of zero curves
`fixedbyzero`:	Price fixed rate note by a set of zero curves
`floatbyzero`:	Price floating rate note by a set of zero curves
`swapbyzero`:	Price swap by a set of zero curves

Each of these functions can be used individually to price an instrument. Consult the reference pages for specific information on the use of these functions.

`intenvprice` takes as input an interest rate term structure created with `intenvset`, and a portfolio of interest rate contingent derivatives instruments created with `instadd`. To learn more about `instadd`, see "Creating and Managing Instrument Portfolios" on page 1-6, and to learn more about the interest rate term structure see "Interest Rate Environment" on page 2-2.

The syntax for using `intenvprice` to price an entire portfolio is

```
Price = intenvprice(RateSpec, InstSet)
```

where:

- `RateSpec` is the interest rate term structure.
- `InstSet` is the name of the portfolio.

Example: Pricing a Portfolio of Instruments

Consider this example of using the `intenvprice` function to price a portfolio of instruments supplied with the Financial Derivatives Toolbox.

The provided MAT-file `deriv.mat` stores a portfolio as an instrument set variable `ZeroInstSet`. The MAT-file also contains the interest rate term structure `ZeroRateSpec`. You can display the instruments with the function `instdisp`.

```
load deriv.mat;
instdisp(ZeroInstSet)

Index Type CouponRate Settle      Maturity     Period Basis...
1    Bond 0.04       01-Jan-2000  01-Jan-2003  1      NaN...
2    Bond 0.04       01-Jan-2000  01-Jan-2004  2      NaN...

Index Type  CouponRate
Settle      Maturity    FixedReset Basis...
3    Fixed 0.04        01-Jan-2000 01-Jan-2003 1          NaN...

Index Type  Spread Settle      Maturity     FloatReset Basis...
4    Float  20     01-Jan-2000  01-Jan-2003  1          NaN...

Index Type LegRate    Settle      Maturity     LegReset Basis...
5    Swap  [0.06 20]  01-Jan-2000  01-Jan-2003  [1  1]  NaN...
```

Use `intenvprice` to calculate the prices for the instruments contained in the portfolio `ZeroInstSet`.

```
format bank
Prices = intenvprice(ZeroRateSpec, ZeroInstSet)
Prices =

        98.72
        97.53
        98.72
       100.55
         3.69
```

The output `Prices` is a vector containing the prices of all the instruments in the portfolio in the order indicated by the `Index` column displayed by `instdisp`. Consequently, the first two elements in `Prices` correspond to the first two bonds; the third element corresponds to the fixed rate note; the fourth to the floating rate note; and the fifth element corresponds to the price of the swap.

Sensitivity

The Financial Derivatives Toolbox can calculate two types of derivative price sensitivities, namely delta and gamma. *Delta* represents the dollar sensitivity of prices to shifts in the observed forward yield curve. *Gamma* represents the dollar sensitivity of delta to shifts in the observed forward yield curve.

The intenvsens function computes instrument sensitivities as well as instrument prices. If you need both the prices and sensitivity measures, use intenvsens. A separate call to intenvprice is not required.

Here is the syntax

```
[Delta, Gamma, Price] = intenvsens(RateSpec, InstSet)
```

where, as before:

- RateSpec is the interest rate term structure.
- InstSet is the name of the portfolio.

Example: Sensitivities and Prices

Here is an example of using intenvsens to calculate both sensitivities and prices.

```
format long
load deriv.mat;
[Delta, Gamma, Price] = intenvsens(ZeroRateSpec, ZeroInstSet);
```

Display the results in a single matrix in long format.

```
All = [Delta Gamma Price]

All =

  1.0e+003 *

  -0.27264034403478   1.02984451401241   0.09871593902758
  -0.34743857788527   1.62265027222659   0.09753338552637
  -0.27264034403478   1.02984451401241   0.09871593902758
  -0.00104445683331   0.00330878190894   0.10055293001355
  -0.28204045553455   1.05962355119047   0.00369230914950
```

To view the per-dollar sensitivity, divide the first two columns by the last one.

```
[Delta./Price, Gamma./Price, Price]

ans =

  1.0e+002 *

  -0.02761867503065    0.10432403562759    0.98715939027581
  -0.03562252822561    0.16636870169834    0.97533385526369
  -0.02761867503065    0.10432403562759    0.98715939027581
  -0.00010387134748    0.00032905872643    1.00552930013547
  -0.76385926561057    2.86981265188338    0.03692309149502
```

Heath-Jarrow-Morton (HJM) Model

The Heath-Jarrow-Morton (HJM) model is one of the most widely used models for pricing interest rate derivatives. The model considers a given initial term structure of interest rates and a specification of the volatility of forward rates to build a tree representing the evolution of the interest rates, based upon a statistical process. For further explanation, see the book *"Modelling Fixed Income Securities and Interest Rate Options"* by Robert A. Jarrow.

Building an HJM Forward Rate Tree

The HJM tree of forward rates is the fundamental unit representing the evolution of interest rates in a given period of time. This section explains how to create the HJM forward rate tree using the Financial Derivatives Toolbox.

The MATLAB function that creates the HJM forward rate tree is `hjmtree`. This function takes three structures as input arguments:

- The volatility model `VolSpec`. (See "Specifying the Volatility Model (VolSpec)" on page 2-22.)
- The interest rate term structure `RateSpec`. (See "Specifying the Interest Rate Term Structure (RateSpec)" on page 2-24.)
- The tree time layout `TimeSpec`. (See "Specifying the Time Structure (TimeSpec)" on page 2-25.)

Creating the HJM Forward Rate Tree (hjmtree)

Calling the function `hjmtree` creates the structure, `HJMTree`, containing time and forward rate information for a bushy tree.

This structure is a self-contained unit that includes the HJM tree of rates (found in the `FwdTree` field of the structure), and the volatility, rate, and time specifications used in building this tree.

The calling syntax for `hjmtree` is

```
HJMTree = hjmtree(VolSpec, RateSpec, TimeSpec)
```

where:

- `VolSpec` is a structure that specifies the forward rate volatility process. `VolSpec` is created using the function `hjmvolspec`. The `hjmvolspec` function supports the specification of multiple factors. It handles five models for the volatility of the interest rate term structure:

  - Constant
  - Stationary
  - Exponential
  - Vasicek
  - Proportional

  Incorporating multiple factors allows you to specify different types of shifts in the shape and location of the interest rate structure. A one-factor model assumes that the interest term structure is affected by a single source of uncertainty.

- `RateSpec` is the interest rate specification of the initial rate curve. This structure is created with the function `intenvset`. (See "Interest Rate Term Structure" on page 2-11.)

- `TimeSpec` is the tree time layout specification. This variable is created with the function `hjmtimespec`. It represents the mapping between level times and level dates for rate quoting. This structure determines indirectly the number of levels of the tree generated in the call to `hjmtree`.

Specifying the Volatility Model (VolSpec)

The function `hjmvolspec` generates the structure `VolSpec`, which specifies the volatility process $\sigma(t, T)$ used in the creation of the forward rate trees. In this context T represents the starting time of the forward rate, and t represents the observation time. The volatility process can be constructed from a combination of factors specified sequentially in the call to `hjmvolspec`. Each factor specification starts with a string specifying the name of the factor, followed by the pertinent parameters.

Consider an example that uses a single factor, specifically, a constant-sigma factor. The constant factor specification requires only one parameter, the value of σ. In this case, the value corresponds to 0.10.

```
VolSpec = hjmvolspec('Constant', 0.10)
```

```
VolSpec =

      FinObj: 'HJMVolSpec'
 FactorModels: {'Constant'}
   FactorArgs: {{1x1 cell}}
   SigmaShift: 0
   NumFactors: 1
    NumBranch: 2
       PBranch: [0.5000 0.5000]
   Fact2Branch: [-1 1]
```

The `NumFactors` field of the `VolSpec` structure, `VolsSpec.NumFactors = 1`, reveals that the number of factors used to generate `VolSpec` was one. The `FactorModels` field indicates that it is a `'Constant'` factor, and the `NumBranches` field indicates the number of branches. As a consequence, each node of the resulting tree has two branches, one going up, and the other going down.

Consider now a two-factor volatility process made from a proportional factor and an exponential factor.

```
% Exponential factor:
Sigma_0 = 0.1;
Lambda = 1;
% Proportional factor
CurveProp = [0.11765; 0.08825; 0.06865];
CurveTerm = [   1  ;    2  ;     3   ];
% Build VolSpec
VolSpec = hjmvolspec('Proportional', CurveProp, CurveTerm,...
1e6,'Exponential', Sigma_0, Lambda)

VolSpec =

      FinObj: 'HJMVolSpec'
 FactorModels: {'Proportional'  'Exponential'}
   FactorArgs: {{1x3 cell}  {1x2 cell}}
   SigmaShift: 0
   NumFactors: 2
    NumBranch: 3
       PBranch: [0.2500 0.2500 0.5000]
   Fact2Branch: [2x3 double]
```

The output shows that the volatility specification was generated using two factors. The tree has three branches per node. Each branch has probabilities of 0.25, 0.25, and 0.5, going from top to bottom.

Specifying the Interest Rate Term Structure (RateSpec)

The structure `RateSpec` is an interest term structure that defines the initial forward rate specification from which the tree rates are derived. The section "Interest Rate Term Structure" on page 2-11 explains how to create these structures using the function `intenvset`, given the interest rates, the starting and ending dates for each rate, and the compounding value.

Consider the example

```
Compounding = 1;
Rates = [0.02; 0.02; 0.02; 0.02];
StartDates = ['01-Jan-2000';
              '01-Jan-2001';
              '01-Jan-2002';
              '01-Jan-2003'];
EndDates =   ['01-Jan-2001';
              '01-Jan-2002';
              '01-Jan-2003';
              '01-Jan-2004'];
ValuationDate = '01-Jan-2000';

RateSpec = intenvset('Compounding',1,'Rates', Rates,...
'StartDates', StartDates, 'EndDates', EndDates,...
'ValuationDate', ValuationDate)

RateSpec =

        FinObj: 'RateSpec'
   Compounding: 1
          Disc: [4x1 double]
         Rates: [4x1 double]
      EndTimes: [4x1 double]
    StartTimes: [4x1 double]
      EndDates: [4x1 double]
    StartDates: [4x1 double]
 ValuationDate: 730486
```

```
        Basis: 0
  EndMonthRule: 1
```

Use the function `datedisp` to examine the dates defined in the variable `RateSpec`. For example

```
datedisp(RateSpec.ValuationDate)
01-Jan-2000
```

Specifying the Time Structure (TimeSpec)

The structure `TimeSpec` specifies the time structure for an HJM tree. This structure defines the mapping between the observation times at each level of the tree and the corresponding dates.

`TimeSpec` is built using the function `hjmtimespec`. The `hjmtimespec` function requires three input arguments:

- The valuation date `ValuationDate`
- The maturity date `Maturity`
- The compounding rate `Compounding`

The syntax used for calling `hjmtimespec` is

```
TimeSpec = hjmtimespec(ValuationDate, Maturity, Compounding)
```

where:

- `ValuationDate` is the first observation date in the tree.
- `Maturity` is a vector of dates representing the cash flow dates of the tree. Any instrument cash flows with these maturities fall on tree nodes.
- `Compounding` is the frequency at which the rates are compounded when annualized.

Calling `hjmtimespec` with the same data used to create the interest rate term structure, `RateSpec` builds the structure that specifies the time layout for the tree.

```
Maturity = EndDates;
TimeSpec = hjmtimespec(ValuationDate, Maturity, Compounding)

TimeSpec =
```

```
          FinObj: 'HJMTimeSpec'
   ValuationDate: 730486
        Maturity: [4x1 double]
     Compounding: 1
           Basis: 0
     EndMonthRule: 1
```

Note that the maturities specified when building `TimeSpec` do not have to coincide with the `EndDates` of the rate intervals in `RateSpec`. Since `TimeSpec` defines the time-date mapping of the HJM tree, the rates in `RateSpec` are interpolated to obtain the initial rates with maturities equal to those found in `TimeSpec`.

Example: Creating an HJM Tree

Use the `VolSpec`, `RateSpec`, and `TimeSpec` you have created as input to the `HJMTree` function to create an HJM tree.

```
% Reset the volatility factor to the Constant case
VolSpec = hjmvolspec('Constant', 0.10);

HJMTree = hjmtree(VolSpec, RateSpec, TimeSpec)

HJMTree =

   FinObj: 'HJMFwdTree'
  VolSpec: [1x1 struct]
 TimeSpec: [1x1 struct]
 RateSpec: [1x1 struct]
     tObs: [0 1 2 3]
     TFwd: {[4x1 double] [3x1 double] [2x1 double] [3]}
   CFlowT: {[4x1 double] [3x1 double] [2x1 double] [4]}
  FwdTree:{[4x1 double][3x1x2 double][2x2x2 double][1x4x2 double]}
```

Using HJM Trees in MATLAB

When working with the HJM model, the Financial Derivatives Toolbox uses trees to represent forward rates, prices, etc. At the highest level, these trees have structures wrapped around them. The structures encapsulate information needed to interpret completely the information contained in a tree.

Consider this example, which uses the interest rate and portfolio data in the MAT-file `deriv.mat` included in the toolbox.

Load the data into the MATLAB workspace.

```
load deriv.mat
```

Display the list of the variables loaded from the MAT-file.

```
whos

Name            Size        Bytes  Class

BDTInstSet      1x1         22708  struct array
BDTTree         1x1          5522  struct array
HJMInstSet      1x1         22700  struct array
HJMTree         1x1          6318  struct array
ZeroInstSet     1x1         14442  struct array
ZeroRateSpec    1x1          1580  struct array
```

Structure of an HJM Tree

You can now examine in some detail the contents of the `HJMTree` structure.

```
HJMTree

HJMTree =

  FinObj: 'HJMFwdTree'
 VolSpec: [1x1 struct]
TimeSpec: [1x1 struct]
RateSpec: [1x1 struct]
    tObs: [0 1 2 3]
    TFwd: {[4x1 double]  [3x1 double]  [2x1 double]  [3]}
  CFlowT: {[4x1 double]  [3x1 double]  [2x1 double]  [4]}
 FwdTree:{[4x1 double][3x1x2 double][2x2x2 double][1x4x2 double]}
```

`FwdTree` contains the actual forward rate tree. It is represented in MATLAB as a cell array with each cell array element containing a tree level.

The other fields contain other information relevant to interpreting the values in `FwdTree`. The most important of these are `VolSpec`, `TimeSpec`, and `RateSpec`,

which contain the volatility, time structure, and rate structure information respectively.

First Node. Observe the forward rates in `FwdTree`. The first node represents the valuation date, `tObs = 0`.

```
HJMTree.FwdTree{1}

ans =

    1.0356
    1.0468
    1.0523
    1.0563
```

Note The Financial Derivatives Toolbox uses *inverse discount* notation for forward rates in the tree. An inverse discount represents a factor by which the present value of an asset is multiplied to find its future value. In general, these forward factors are reciprocals of the discount factors.

Look closely at the `RateSpec` structure used in generating this tree to see where these values originate. Arrange the values in a single array.

```
[HJMTree.RateSpec.StartTimes HJMTree.RateSpec.EndTimes...
HJMTree.RateSpec.Rates]

ans =

         0    1.0000    0.0356
    1.0000    2.0000    0.0468
    2.0000    3.0000    0.0523
    3.0000    4.0000    0.0563
```

If you find the corresponding inverse discounts of the interest rates in the third column, you have the values at the first node of the tree. You can turn interest rates into inverse discounts using the function `rate2disc`.

```
Disc = rate2disc(HJMTree.TimeSpec.Compounding,...
HJMTree.RateSpec.Rates, HJMTree.RateSpec.EndTimes,...
HJMTree.RateSpec.StartTimes);
```

```
FRates = 1./Disc

FRates =
     1.0356
     1.0468
     1.0523
     1.0563
```

Second Node. The second node represents the first rate observation time, `tObs` = 1. This node displays two states: one representing the branch going up and the other representing the branch going down.

Note that `HJMTree.VolSpec.NumBranch = 2`.

```
HJMTree.VolSpec

ans =

          FinObj: 'HJMVolSpec'
     FactorModels: {'Constant'}
       FactorArgs: {{1x1 cell}}
       SigmaShift: 0
       NumFactors: 1
        NumBranch: 2
          PBranch: [0.5000 0.5000]
       Fact2Branch: [-1 1]
```

Examine the rates of the node corresponding to the up branch.

```
HJMTree.FwdTree{2}(:,:,1)

ans =

     1.0364
     1.0420
     1.0461
```

Now examine the corresponding down branch.

```
HJMTree.FwdTree{2}(:,:,2)

ans =
```

```
                 1.0574
                 1.0631
                 1.0672
```

Third Node. The third node represents the second observation time, tObs = 2. This node contains a total of four states, two representing the branches going up and the other two representing the branches going down.

Examine the rates of the node corresponding to the up states.

```
    HJMTree.FwdTree{3}(:,:,1)

    ans =

         1.0317    1.0526
         1.0358    1.0568
```

Next examine the corresponding down states.

```
    HJMTree.FwdTree{3}(:,:,2)

    ans =

         1.0526    1.0738
         1.0568    1.0781
```

Isolating a Specific Node. Starting at the third level, indexing within the tree cell array becomes complex, and isolating a specific node can be difficult. The function bushpath isolates a specific node by specifying the path to the node as a vector of branches taken to reach that node. As an example, consider the node reached by starting from the root node, taking the branch up, then the branch down, and then another branch down. Given that the tree has only two branches per node, branches going up correspond to a 1, and branches going down correspond to a 2. The path up-down-down becomes the vector [1 2 2].

```
    FRates = bushpath(HJMTree.FwdTree, [1 2 2])

    FRates =

         1.0356
         1.0364
```

```
     1.0526
     1.0674
```

`bushpath` returns the spot rates for all the nodes touched by the path specified in the input argument, the first one corresponding to the root node, and the last one corresponding to the target node.

Isolating the same node using direct indexing obtains

```
HJMTree.FwdTree{4}(:, 3, 2)

ans =

     1.0674
```

As expected, this single value corresponds to the last element of the rates returned by `bushpath`.

You can use these techniques with any type of tree generated with the Financial Derivatives Toolbox, such as forward rate trees or price trees.

Graphical View of Forward Rate Tree

The function `treeviewer` provides a graphical view of the path of forward rates specified in `HJMTree`. For example, here is a `treeviewer` representation of the rates along both the up and the down branches of `HJMTree`.

```
treeviewer(HJMTree)
```
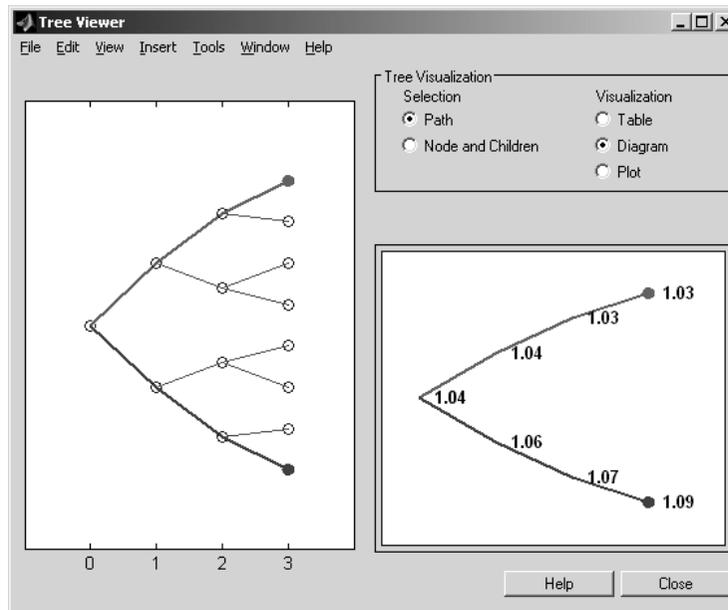
A previous example used bushpath to find the path of forward rates taking the first branch up and then two branches down the rate tree.

```
FRates = bushpath(HJMTree.FwdTree, [1 2 2])

FRates =

    1.0356
    1.0364
    1.0526
    1.0674
```

The treeviewer function displays the same information obtained by clicking along the sequence of nodes, as shown next.

Note If you are not familiar with the use of treeviewer, see the treeviewer reference page for complete instructions on its use.
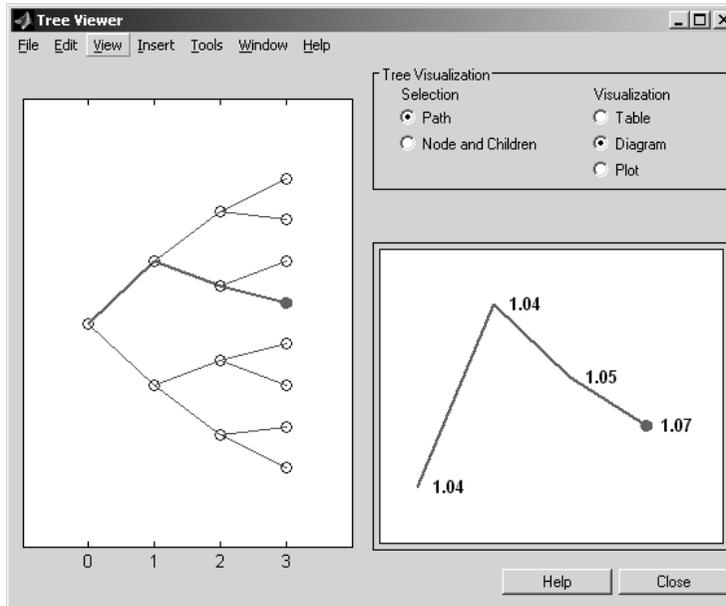
Pricing and Sensitivity from HJM

This section explains how to use the Financial Derivatives Toolbox to compute prices and sensitivities of several financial instruments using the Heath-Jarrow-Morton (HJM) model. For information, see:

- "Pricing and the Price Tree" on page 2-34 for a discussion of using the `hjmprice` function to compute prices for a portfolio of instruments.
- "Calculating Prices and Sensitivities" on page 2-50 for a discussion of using the `hjmsens` function to compute delta, gamma, and vega portfolio sensitivities.

Pricing and the Price Tree

For the HJM model, the function `hjmprice` calculates the price of any set of supported instruments, based on an interest rate tree. The function is capable of pricing these instrument types:

- Bonds
- Bond options
- Arbitrary cash flows
- Fixed-rate notes
- Floating-rate notes
- Caps
- Floors
- Swaps

The syntax used for calling `hjmprice` is

```
[Price, PriceTree] = hjmprice(HJMTree, InstSet, Options)
```

This function requires two input arguments: the interest rate tree, `HJMTree`, and the set of instruments, `InstSet`. An optional argument `Options` further controls the pricing and the output displayed.

`HJMTree` is the Heath-Jarrow-Morton tree sampling of a forward rate process, created using `hjmtree`. See "Building an HJM Forward Rate Tree" on page 2-21 to learn how to create this structure.

InstSet is the set of instruments to be priced. This structure represents the set of instruments to be priced independently using the HJM model. The section "Creating and Managing Instrument Portfolios" on page 1-6 explains how to create this variable.

Options is an options structure created with the function derivset. This structure defines how the HJM tree is used to find the price of instruments in the portfolio, and how much additional information is displayed in the command window when calling the pricing function. If this input argument is not specified in the call to hjmprice, a default Options structure is used.

hjmprice classifies the instruments and calls the appropriate pricing function for each one of the instrument types. The pricing functions are bondbyhjm, cfbyhjm, fixedbyhjm, floatbyhjm, optbndbyhjm, and swapbyhjm. You can also use these functions directly to calculate the price of sets of instruments of the same type. See the documentation for these individual functions for further information.

Example: HJM Pricing

Consider the following example, which uses the portfolio and interest rate data in the MAT-file deriv.mat included in the toolbox. Load the data into the MATLAB workspace.

```
load deriv.mat
```

Use the MATLAB whos command to display a list of the variables loaded from the MAT-file.

```
whos

  Name               Size          Bytes  Class

    BDTInstSet         1x1           22708  struct array
    BDTTree            1x1            5522  struct array
    HJMInstSet         1x1           22700  struct array
    HJMTree            1x1            6318  struct array
    ZeroInstSet        1x1           14442  struct array
    ZeroRateSpec       1x1            1580  struct array
```

HJMTree and HJMInstSet are the input arguments needed to call the function hjmprice.

Use the function `instdisp` to examine the set of instruments contained in the variable `HJMInstSet`.

```
instdisp(HJMInstSet)
```

```
Index Type CouponRate Settle      Maturity    Period Basis ......Name      Quantity
1     Bond 0.04       01-Jan-2000 01-Jan-2003 1      NaN.........4% bond   100
2     Bond 0.04       01-Jan-2000 01-Jan-2004 2      NaN.........4% bond   50

Index Type    UnderInd OptSpec Strike ExerciseDates AmericanOpt Name       Quantity
3     OptBond 2        call    101    01-Jan-2003   NaN         Option 101 -50

Index Type  CouponRate Settle      Maturity    FixedReset Basis Principal Name     Quantity
4     Fixed 0.04       01-Jan-2000 01-Jan-2003 1          NaN   NaN       4% Fixed 80

Index Type  Spread Settle      Maturity    FloatReset Basis Principal Name       Quantity
5     Float 20     01-Jan-2000 01-Jan-2003 1          NaN   NaN       20BP Float 8

Index Type Strike Settle      Maturity    CapReset Basis Principal Name   Quantity
6     Cap  0.03   01-Jan-2000 01-Jan-2004 1        NaN   NaN       3% Cap 30

Index Type  Strike Settle      Maturity    FloorReset Basis Principal Name     Quantity
7     Floor 0.03   01-Jan-2000 01-Jan-2004 1          NaN   NaN       3% Floor 40

Index Type LegRate     Settle      Maturity    LegReset Basis Principal LegType Name       Quantity
8     Swap [0.06 20] 01-Jan-2000 01-Jan-2003 [1  1]   NaN   NaN       [NaN]   6%/20BP Swap 10
```

Note that there are eight instruments in this portfolio set: two bonds, one bond option, one fixed rate note, one floating rate note, one cap, one floor, and one swap. Each instrument has a corresponding index that identifies the instrument prices in the price vector returned by `hjmprice`.

Now use `hjmprice` to calculate the price of each instrument in the instrument set.

```
Price = hjmprice(HJMTree, HJMInstSet)
Warning: Not all cash flows are aligned with the tree. Result will
be approximated.

Price =

    98.7159
    97.5280
     0.0486
    98.7159
   100.5529
```

```
6.2831
0.0486
3.6923
```

Note The warning shown above appears because some of the cash flows for the second bond do not fall exactly on a tree node. This situation is discussed further in "HJM Pricing Options Structure" on page 2-44.

Price Vector

The prices in the output vector `Price` correspond to the prices at observation time zero (`tObs = 0`), which is defined as the valuation date of the interest rate tree. The instrument indexing within `Price` is the same as the indexing within `InstSet`. In this example, the prices in the `Price` vector correspond to the instruments in the following order.

```
InstNames = instget(HJMInstSet, 'FieldName','Name')

InstNames =

4% bond
4% bond
Option 101
4% Fixed
20BP Float
3% Cap
3% Floor
6%/20BP Swap
```

Consequently, in the `Price` vector, the fourth element, `98.7159`, represents the price of the fourth instrument (4% fixed-rate note); the sixth element, `6.2831`, represents the price of the sixth instrument (3% cap).

Price Tree Structure

If you call the `hjmprice` function with two output arguments, e.g.,

```
[Price, PriceTree] = hjmprice(HJMTree, HJMInstSet)
```

you generate a price tree along with the price information.

The optional output price tree structure `PriceTree` holds all the pricing information. The first field of this structure, `FinObj`, indicates that this structure represents a price tree. The second field, `PBush` is the tree holding the price of the instruments in each node of the tree. The third field, `AIBush` is the tree holding the accrued interest of the instruments in each node of the tree. Finally, the fourth field, `tObs`, represents the observation time of each level of `PBush` and `AIBush`, with units in terms of compounding periods.

In this example the price tree looks like

```
PriceTree =

FinObj: 'HJMPriceTree'
 PBush: {[8x1 double]  [8x1x2 double]  ...[8x8 double]}
AIBush: {[8x1 double]  [8x1x2 double] ... [8x8 double]}
  tObs: [0 1 2 3 4]
```

Both `PBush` and `AIBush` are actually 1-by-5 cell arrays, consistent with the five observation times of `tObs`. The data display has been shortened here to fit on a single line.

Using the command line interface, you can directly examine `PriceTree.PBush`, the field within the `PriceTree` structure that contains the price tree with the price vectors at every state. The first node represents `tObs = 0`, corresponding to the valuation date.

```
PriceTree.PBush{1}

ans =

    98.7159
    97.5280
     0.0486
    98.7159
   100.5529
     6.2831
     0.0486
     3.6923
```

With this interface you can observe the prices for *all* instruments in the portfolio at *a specific time*.

Using treeviewer to View Instrument Prices Through Time

You can use the function `treeviewer` to display a graphical representation of the tree, allowing you to examine interactively the prices on the nodes of the tree until maturity. With `treeviewer` you select *each instrument individually* in the instrument portfolio for display. For example, the command

```
treeviewer(PriceTree, HJMInstSet)
```

with the 4% bond instrument selected in the **Instrument** window shows prices for this bond instrument along the price tree from the valuation date until maturity.
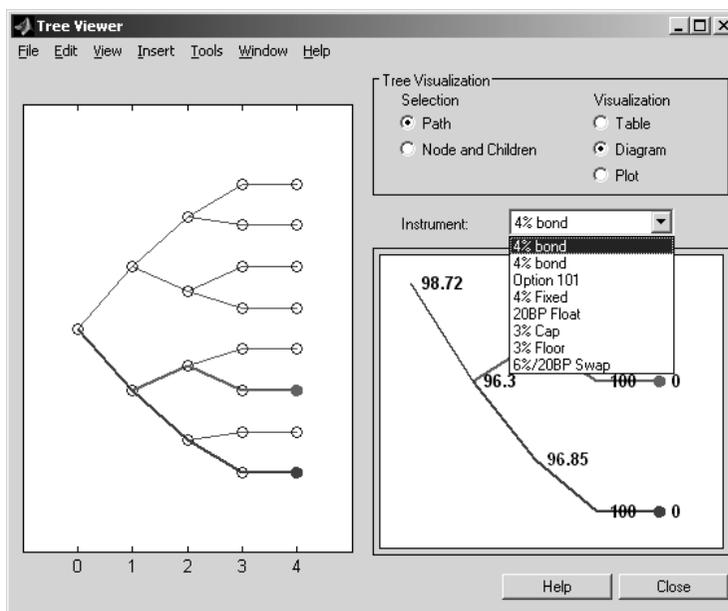
Example: Valuation Date Prices

You can use `treeviewer` instrument-by-instrument to observe instrument prices through time. For the first 4% bond in the instrument portfolio, `treeviewer` indicates a valuation date price of 98.72, the same value obtained by accessing the `PriceTree` structure directly.
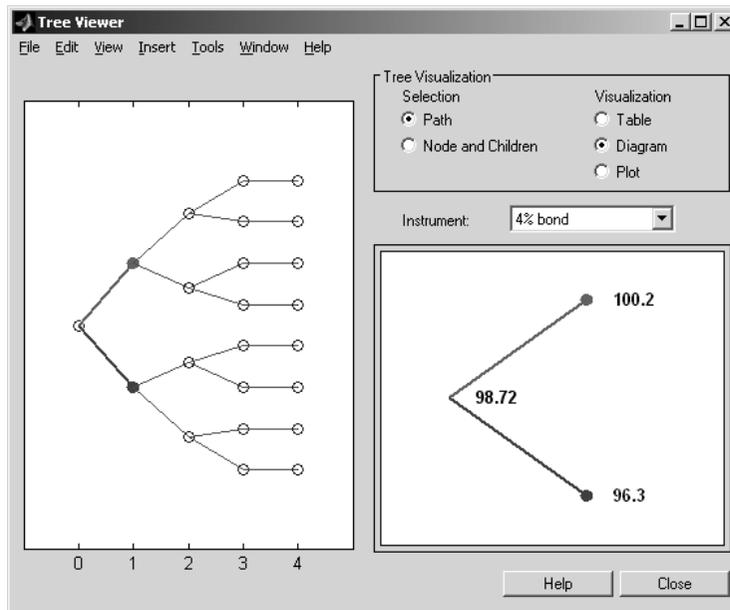
As a further example, look at the sixth instrument in the price vector, the 3% cap. At the valuation date its value obtained directly from the structure is 6.2831. Use `treeviewer` on this instrument to confirm this price.
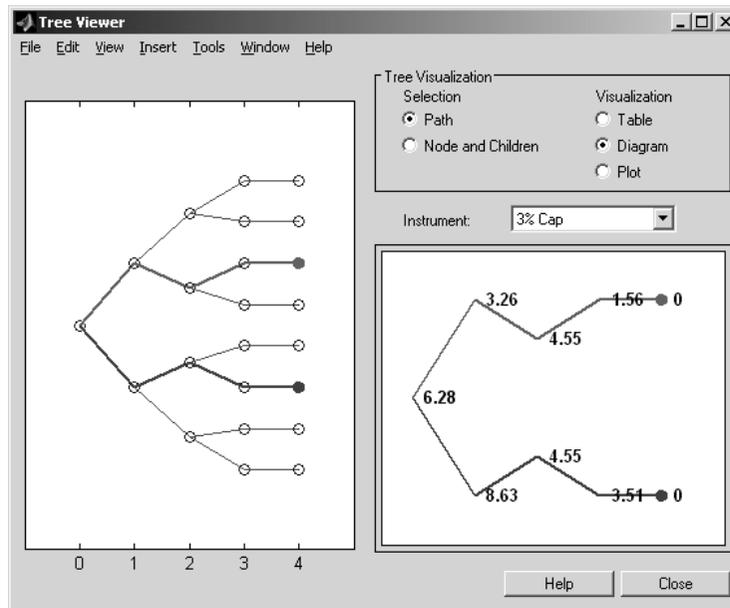
Example: Additional Observation Times

The second node represents the first rate observation time, tObs = 1. This node displays two states, one representing the branch going up and the other one representing the branch going down.

Examine the prices of the node corresponding to the up branch.

```
PriceTree.PBush{2}(:,:,1)

ans =

  100.1563
   99.7309
    0.1007
  100.1563
  100.3782
    3.2594
    0.1007
    3.5597
```

As before, you can use `treeviewer`, this time to examine the price for the 4% bond on the up branch. `treeviewer` displays a price of 100.2 for the first node of the up branch, as expected.
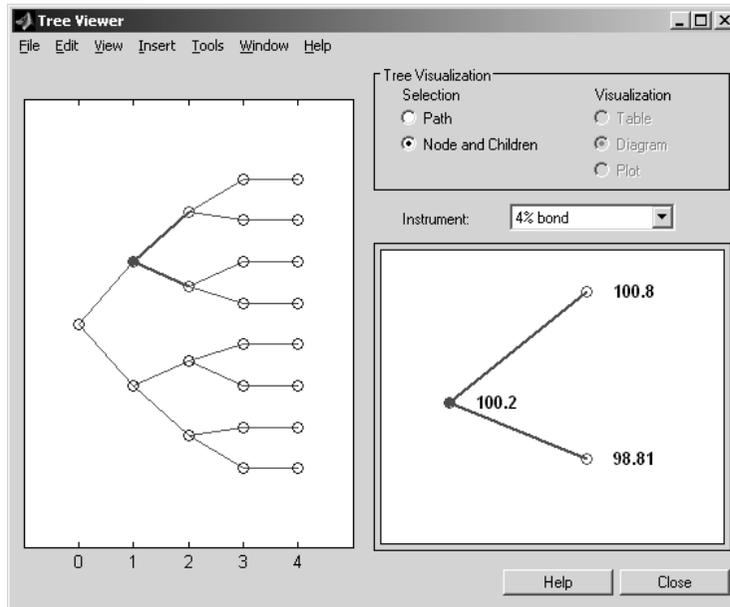
Now examine the corresponding down branch.

```
PriceTree.PBush{2}(:,:,2)

ans =

    96.3041
    94.1986
         0
    96.3041
   100.3671
     8.6342
         0
    -0.3923
```

Use `treeviewer` once again, now to observe the price of the 4% bond on the down branch. The displayed price of 96.3 conforms to the price obtained from

direct access of the `PriceTree` structure. You may continue this process as far along the price tree as you want.
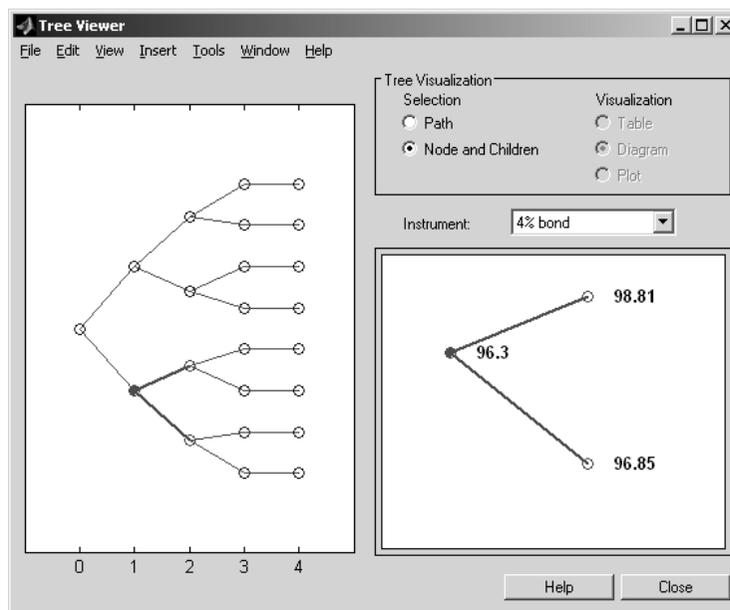
HJM Pricing Options Structure

The MATLAB structure `Options` provides additional input to each pricing function. The `Options` structure

• Tells pricing functions how to use the interest rate tree to calculate instrument prices

• Determines what additional information the command window displays along with instrument prices.

You provide pricing options in an optional `Options` argument passed to each pricing function. (See, for example, `bondbyhjm` or `hjmprice`.)

Default Structure

If you do not specify the `Options` argument in the call to a pricing function, the function uses a default structure. To observe the default structure, use `derivset` without any arguments.

```
Options = derivset

Options =

    Diagnostics: 'off'
       Warnings: 'on'
       ConstRate: 'on'
```

The `Options` structure has three fields: `Diagnostics`, `Warnings`, and `ConstRate`.

`Diagnostics` indicates whether additional information is displayed if the HJM tree is modified. The default value for this option is `'off'`. If `Diagnostics` is set to `'on'` and `ConstRate` is set to `'off'`, the pricing functions display information such as the number of nodes in the last level of the HJM tree generated for pricing purposes.

`Warnings` indicates whether to display warning messages when the input tree is not adequate for accurately pricing the instruments. The default value for this option is `'on'`. If both `ConstRate` and `Warnings` are `'on'`, a warning is displayed if any of the instruments in the input portfolio has a cash flow date between tree dates. If `ConstRate` is `'off'`, and `Warnings` is `'on'`, a warning is displayed if the tree is modified to match the cash flow dates on the instruments in the portfolio.

`ConstRate` indicates whether the interest rates should be assumed constant between tree dates. By default this option is `'on'`, which is not an arbitrage-free assumption. Consequently the pricing functions return an approximate price for instruments featuring cash flows between tree dates. Instruments featuring cash flows only on tree nodes are not affected by this option and return exact (arbitrage-free) prices. When `ConstRate` is `'off'`, the HJM pricing function finds the cash flow dates for all instruments in the portfolio. If these cash flows do not align exactly with the tree dates, a new tree is generated and used for pricing. This new tree features the same volatility and initial rate specifications of the input HJM tree but contains tree nodes for each date in which at least one instrument in the portfolio has a cash flow. Keep in mind that the number of nodes in an HJM tree grows exponentially with the number of tree dates. Consequently, setting `ConstRate` `'off'` dramatically increases the memory and CPU demands on the computer.

Customizing the Structure

Customize the `Options` structure by passing property name/property value pairs to the `derivset` function.

As an example, consider an `Options` structure with `ConstRate` `'off'` and `Diagonistics` `'on'`.

```
Options = derivset('ConstRate', 'off', 'Diagnostics', 'on')

Options =

Diagnostics: 'on'
   Warnings: 'on'
  ConstRate: 'off'
```

To obtain the value of a specific property from the `Options` structure, use `derivget`.

```
CR = derivget(Options, 'ConstRate')

CR =
Off
```

Note Use `derivset` and `derivget` to construct the `Options` structure. These functions are guaranteed to remain unchanged, while the implementation of the structure itself may be modified in the future.

Now observe the effects of setting `ConstRate` `'off'`. Obtain the tree dates from the HJM tree.

```
TreeDates = [HJMTree.TimeSpec.ValuationDate;...
HJMTree.TimeSpec.Maturity]

TreeDates =

    730486
    730852
    731217
    731582
    731947
```

```
datedisp(TreeDates)

01-Jan-2000
01-Jan-2001
01-Jan-2002
01-Jan-2003
01-Jan-2004
```

All instruments in `HJMInstSet` settle on Jan 1st, 2000, and all have cash flows once a year, with the exception of the second bond, which features a period of 2. This bond has cash flows twice a year, with every other cash flow consequently falling between tree dates. You can extract this bond from the portfolio to compare how its price differs by setting `ConstRate` to `'on'` and `'off'`.

```
BondPort = instselect(HJMInstSet, 'Index', 2);

instdisp(BondPort)

Index Type CouponRate Settle      Maturity    Period Basis...
1     Bond 0.04       01-Jan-2000 01-Jan-2004 2      NaN...
```

First price the bond with `ConstRate` `'on'` (default).

```
format long
[BondPrice, BondPriceTree] = hjmprice(HJMTree, BondPort)
Warning: Not all cash flows are aligned with the tree. Result will
be approximated.

BondPrice =

  97.52801411736377

BondPriceTree =
FinObj: 'HJMPriceTree'
 PBush: {1x5 cell}
AIBush: {[0]  [1x1x2 double] ... [1x4x2 double]  [1x8 double]}
  tObs: [0 1 2 3 4]
```

Now recalculate the price of the bond setting `ConstRate` `'off'`.

```
OptionsNoCR = derivset('ConstR', 'off')
```

```
OptionsNoCR =

Diagnostics: 'off'
   Warnings: 'on'
  ConstRate: 'off'

[BondPriceNoCR, BondPriceTreeNoCR] = hjmprice(HJMTree,...
BondPort, OptionsNoCR)
Warning: Not all cash flows are aligned with the tree. Rebuilding
tree.

BondPriceNoCR =

  97.53342361674437

BondPriceTreeNoCR =

FinObj: 'HJMPriceTree'
 PBush: {1x9 cell}
AIBush: {1x9 cell}
  tObs: [0 0.5000 1 1.5000 2 2.5000 3 3.5000 4]
```

As indicated in the last warning, because the cash flows of the bond did not
align with the tree dates, a new tree was generated for pricing the bond. This
pricing method returns more accurate results since it guarantees that the
process is arbitrage-free. It also takes longer to calculate and requires more
memory. The tObs field of the price tree structure indicates the increased
memory usage. BondPriceTree.tObs has only five elements, while
BondPriceTreeNoCR.tObs has nine. While this may not seem like a large
difference, it has a dramatic effect on the number of states in the last node.

```
size(BondPriceTree.PBush{end})

ans =

     1 8

size(BondPriceTreeNoCR.PBush{end})

ans =
```

```
1 128
```

The differences become more obvious by examining the price trees with `treeviewer`.

```
treeviewer(BondPriceTree, BondPort)
```
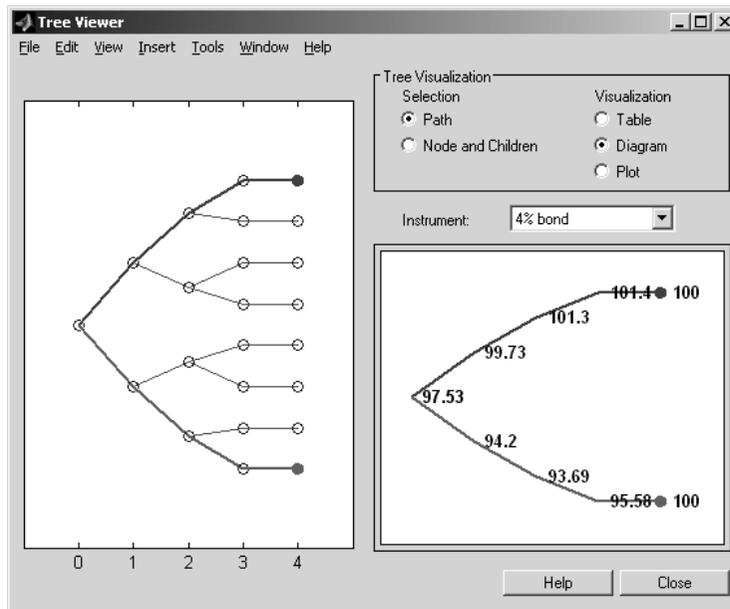


```
treeviewer(BondPriceTreeNoCR, BondPort)
```
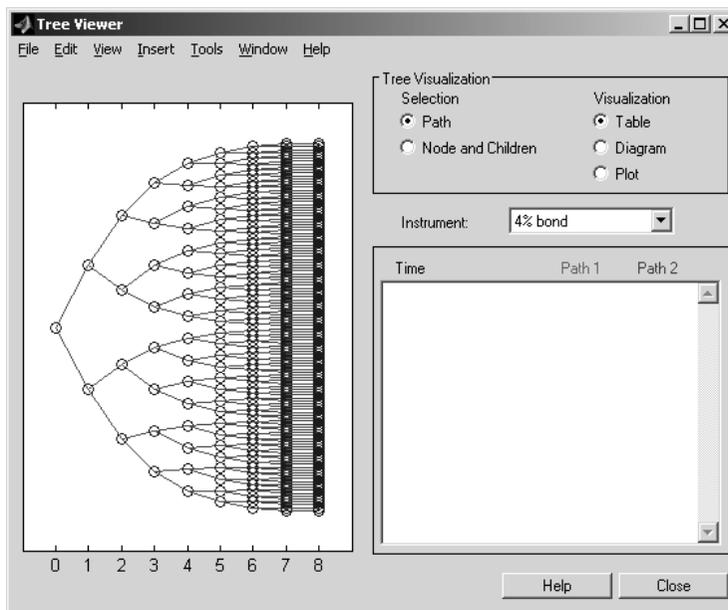
Calculating Prices and Sensitivities

The function hjmsens computes the delta, gamma, and vega sensitivities of instruments using an interest rate tree created with hjmtree. It also optionally returns the calculated price for each instrument. hjmsens requires the same two input arguments used by hjmprice, namely HJMTree and HJMInstSet.

hjmsens calculates the dollar value of delta and gamma by shifting the observed forward yield curve by 100 basis points in each direction, and the dollar value of vega by shifting the volatility process by 1%. To obtain the per-dollar value of the sensitivities, divide the dollar sensitivity by the price of the corresponding instrument.

The calling syntax for the function is

```
[Delta, Gamma, Vega, Price] = hjmsens(HJMTree, HJMInstSet)
```

Use the previous example data to calculate the price of instruments.

```
load deriv.mat
[Delta, Gamma, Vega, Price] = hjmsens(HJMTree, HJMInstSet);
```

```
Warning: Not all cash flows are aligned with the tree. Result will
be approximated.
```

Note The warning appears because some of the cash flows for the second
bond do not fall exactly on a tree node. This situation was discussed in "HJM
Pricing Options Structure" on page 2-44.

You can conveniently examine the sensitivities and the prices by arranging
them into a single matrix.

```
All = [Delta, Gamma, Vega, Price]

All =

        -272.65        1029.90           0.00          98.72
        -347.43        1622.69          -0.04          97.53
          -8.08         643.40          34.07           0.05
        -272.65        1029.90           0.00          98.72
          -1.04           3.31              0         100.55
         294.97        6852.56          93.69           6.28
         -47.16        8459.99          93.69           0.05
        -282.05        1059.68           0.00           3.69
```

As with the prices, each row of the sensitivity vectors corresponds to the
similarly indexed instrument in HJMInstSet. To view the *per-dollar
sensitivities*, divide each dollar sensitivity by the corresponding instrument
price.

```
All = [Delta ./ Price, Gamma ./ Price, Vega ./ Price, Price]

All =

         -2.76          10.43          0.00         98.72
         -3.56          16.64         -0.00         97.53
       -166.18       13235.59        700.96          0.05
         -2.76          10.43          0.00         98.72
         -0.01           0.03             0        100.55
         46.95        1090.63         14.91          6.28
       -969.85      173969.77       1926.72          0.05
        -76.39         287.00          0.00         3.690
```

Black-Derman-Toy Model (BDT)

The Black-Derman-Toy (BDT) model is an analytical model used for pricing interest rate derivatives. The model considers a given initial zero rate term structure of interest rates and a specification of the yield volatilities of long rates to build a tree representing the evolution of the interest rates. For further explanation, see the paper "A One Factor Model of Interest Rates and its Application to Treasury Bond Options" by Fischer Black, Emanuel Derman, and William Toy.

Building a BDT Interest Rate Tree

The BDT interest rate tree represents the evolution of interest rates in a given period of time. This section explains how to create the BDT interest rate tree using the Financial Derivatives Toolbox.

The MATLAB function that creates the BDT interest rate tree is `bdttree`. This function takes three structures as input arguments:

- The volatility model `VolSpec`. (See "Specifying the Volatility (VolSpec)" on page 2-54.)
- The interest rate term structure `RateSpec`. (See "Specifying the Interest Rate Term Structure (RateSpec)" on page 2-55.)
- The tree time layout `TimeSpec`. (See "Specifying the Time Structure (TimeSpec)" on page 2-56.)

Creating the BDT Interest Rate Tree (bdttree)

Calling the function `bdttree` creates the structure, `BDTTree`, containing time and interest rate information of the recombining tree.

This structure is a self-contained unit that includes the BDT tree of rates (found in the `FwdTree` field), and the volatility, rate, and time specifications used in building this tree.

The calling syntax for `bdttree` is

```
BDTTree = bdttree(VolSpec, RateSpec, TimeSpec)
```

where:

- `VolSpec` is a structure that specifies the interest rate volatility process. `VolSpec` is created using the function `bdtvolspec`.

- `RateSpec` is the interest rate specification of the initial rate curve. This structure is created with the function `intenvset`. (See "Interest Rate Term Structure" on page 2-11.)

- `TimeSpec` is the tree time layout specification. This variable is created with the function `bdttimespec`. It represents the mapping between level times and level dates for rate quoting. This structure determines indirectly the number of levels of the tree generated in the call to `bdttree`.

Specifying the Volatility (VolSpec)

The function `bdtvolspec` generates the structure `VolSpec`, which specifies the volatility process. The function requires three input arguments:

- The valuation date `ValuationDate`
- The yield volatility end dates `VolDates`
- The yield volatility values `VolCurve`

An optional fourth argument `InterpMethod`, specifying the interpolation method, can be included.

The syntax used for calling `bdtvolspec` is

```
VolSpec = bdtvolspec(ValuationDate, VolDates, VolCurve,...
InterpMethod)
```

where:

- `ValuationDate` is the first observation date in the tree.
- `VolDates` is a vector of dates representing yield volatility end dates.
- `VolCurve` is a vector of yield volatility values.
- `InterpMethod` is the method of interpolation to use. The default is `'linear'`.

Consider the example

```
ValuationDate = datenum('01-01-2000');
EndDates = datenum(['01-01-2001'; '01-01-2002'; '01-01-2003';
'01-01-2004'; '01-01-2005']);
Volatility = [.2; .19; .18; .17; .16];
```

Use `bdtvolspec` to create a volatility specification. Because no interpolation method is explicitly specified, the function uses the `'linear'` default.

```
BDTVolSpec = bdtvolspec(ValuationDate, EndDates, Volatility)

BDTVolSpec =
                 FinObj: 'BDTVolSpec'
          ValuationDate: 730486
                VolDates: [5x1 double]
                VolCurve: [5x1 double]
        VolInterpMethod: 'linear'
```

Specifying the Interest Rate Term Structure (RateSpec)

The structure RateSpec is an interest term structure that defines the initial
interest rate specification from which the tree rates are derived. The section
"Interest Rate Term Structure" on page 2-11 explains how to create these
structures using the function intenvset, given the interest rates, the starting
and ending dates for each rate, and the compounding value.

Consider the example

```
Compounding = 1;
Rates = [0.02; 0.02; 0.02; 0.02];
StartDates = ['01-Jan-2000';
              '01-Jan-2001';
              '01-Jan-2002';
              '01-Jan-2003'];
EndDates =   ['01-Jan-2001';
              '01-Jan-2002';
              '01-Jan-2003';
              '01-Jan-2004'];
ValuationDate = '01-Jan-2000';

RateSpec = intenvset('Compounding',1,'Rates', Rates,...
'StartDates', StartDates, 'EndDates', EndDates,...
'ValuationDate', ValuationDate)

RateSpec =

        FinObj: 'RateSpec'
    Compounding: 1
           Disc: [4x1 double]
          Rates: [4x1 double]
```

```
    EndTimes: [4x1 double]
  StartTimes: [4x1 double]
    EndDates: [4x1 double]
  StartDates: [4x1 double]
ValuationDate: 730486
       Basis: 0
 EndMonthRule: 1
```

Use the function `datedisp` to examine the dates defined in the variable `RateSpec`. For example,

```
datedisp(RateSpec.ValuationDate)
01-Jan-2000
```

Specifying the Time Structure (TimeSpec)

The structure `TimeSpec` specifies the time structure for a BDT tree. This structure defines the mapping between the observation times at each level of the tree and the corresponding dates.

`TimeSpec` is built using the function `bdttimespec`. The `bdttimespec` function requires three input arguments:

- The valuation date `ValuationDate`
- The maturity date `Maturity`
- The compounding rate `Compounding`

The syntax used for calling `bdttimespec` is

```
TimeSpec = bdttimespec(ValuationDate, Maturity, Compounding)
```

where:

- `ValuationDate` is the first observation date in the tree.
- `Maturity` is a vector of dates representing the cash flow dates of the tree. Any instrument cash flows with these maturities fall on tree nodes.
- `Compounding` is the frequency at which the rates are compounded when annualized.

Calling `bdttimespec` with the same data used to create the interest rate term structure, `RateSpec` builds the structure that specifies the time layout for the tree.

```
        Maturity = EndDates;
        TimeSpec = bdttimespec(ValuationDate, Maturity, Compounding)

        TimeSpec =

                FinObj: 'BDTTimeSpec'
        ValuationDate: 730486
             Maturity: [4x1 double]
          Compounding: 1
                Basis: 0
          EndMonthRule: 1
```

Note that the maturities specified when building `TimeSpec` do not have to coincide with the `EndDates` of the rate intervals in `RateSpec`. Since `TimeSpec` defines the time-date mapping of the BDT tree, the rates in `RateSpec` are interpolated to obtain the initial rates with maturities equal to those found in `TimeSpec`.

Example: Creating a BDT Tree

Use the previously computed values for `VolSpec`, `RateSpec`, and `TimeSpec` as input to the function `bdttree` to create a BDT Tree.

```
  BDTTree = bdttree(BDTVolSpec, RateSpec, TimeSpec)

  BDTTree =

    FinObj: 'BDTFwdTree'
    VolSpec: [1x1 struct]
   TimeSpec: [1x1 struct]
   RateSpec: [1x1 struct]
       tObs: [0 1.00 2.00 3.00]
       TFwd: {[4x1 double]  [3x1 double]  [2x1 double]  [3.00]}
      CFlowT: {[4x1 double]  [3x1 double]  [2x1 double]  [4.00]}
    FwdTree: {[1.02] [1.02 1.02] [1.01 1.02 1.03] [1.01 1.02 1.02 1.03]}
```

Using BDT Trees in MATLAB

When working with the BDT model, the Financial Derivatives Toolbox uses trees to represent interest rates, prices, etc. At the highest level, these trees contain several MATLAB structures. The structures encapsulate information needed to interpret completely the information contained in a tree.

Because BDT trees are essentially MATLAB structures, you can examine their contents manually, just as you can for HJM trees. Consider this example, which uses the data in the MAT-file `deriv.mat` included in the toolbox.

Load the data into the MATLAB workspace.

```
load deriv.mat
```

Display the list of the variables loaded from the MAT-file.

```
whos

Name              Size          Bytes  Class

BDTInstSet        1x1           22708  struct array
BDTTree           1x1            5522  struct array
HJMInstSet        1x1           22700  struct array
HJMTree           1x1            6318  struct array
ZeroInstSet       1x1           14442  struct array
ZeroRateSpec      1x1            1580  struct array
```

Structure of a BDT Tree

You can now examine in some detail the contents of the BDTTree structure.

```
BDTTree

BDTTree =

     FinObj: 'BDTFwdTree'
    VolSpec: [1x1 struct]
   TimeSpec: [1x1 struct]
   RateSpec: [1x1 struct]
       tObs: [0 1 2 3]
       TFwd: {[4x1 double]  [3x1 double]  [2x1 double]  [3]}
      CFlowT: {[4x1 double]  [3x1 double]  [2x1 double]  [4]}
     FwdTree: {1x4 cell}
```

2-57

`FwdTree` contains the actual rate tree. It is represented in MATLAB as a cell array with each cell array element containing a tree level.

The other fields contain other information relevant to interpreting the values in `FwdTree`. The most important of these are `VolSpec`, `TimeSpec`, and `RateSpec`, which contain the volatility, rate structure, and time structure information respectively.

Look at the `RateSpec` structure used in generating this tree to see where these values originate. Arrange the values in a single array.

```
[BDTTree.RateSpec.StartTimes BDTTree.RateSpec.EndTimes...
BDTTree.RateSpec.Rates]

ans =

        0    1.0000    0.1000
        0    2.0000    0.1100
        0    3.0000    0.1200
        0    4.0000    0.1250
```

Note The Financial Derivatives Toolbox uses *inverse discount* notation for forward rates in the tree. An inverse discount represents a factor by which the present value of an asset is multiplied to find its future value. In general, these forward factors are reciprocals of the discount factors.

Look at the rates in `FwdTree`. The first node represents the valuation date, `tObs = 0`. The second node represents `tObs = 1`. Examine the rates at the second, third and fourth nodes.

```
BDTTree.FwdTree{2}

ans =

      1.0979    1.1432
```

The second node represents the first observation time, `tObs = 1`. This node contains a total of two states, one representing the branch going up (`1.0979`) and the other representing the branch going down (`1.1432`).

Note The convention in this document is to display *prices* going up on the upper branch. Consequently, when displaying *rates*, rates are falling on the upper branch and increasing on the lower.

```
BDTTree.FwdTree{3}

ans =

    1.0976    1.1377    1.1942
```

The third node represents the second observation time, tObs = 2. This node contains a total of three states, one representing the branch going up (1.0976), one representing the branch in the middle (1.1377) and the other representing the branch going down (1.1942).

```
BDTTree.FwdTree{4}

ans =

    1.0872    1.1183    1.1606    1.2179
```

The fourth node represents the third observation time, tObs = 3. This node contains a total of four states, one representing the branch going up (1.0872), two representing the branches in the middle (1.1183 and 1.1606) and the other representing the branch going down (1.2179).

Verifying Results with treepath

The function treepath isolates a specific node by specifying the path to the node as a vector of branches taken to reach that node. As an example, consider the node reached by starting from the root node, taking the branch up, then the branch down, and finally another branch down. Given that the tree has only two branches per node, branches going up correspond to a 1, and branches going down correspond to a 2. The path up-down-down becomes the vector [1 2 2].

```
  FRates = treepath(BDTTree.FwdTree, [1 2 2])

  FRates =
```

```
          1.1000
          1.0979
          1.1377
          1.1606
```

treepath returns the short rates for all the nodes touched by the path specified in the input argument, the first one corresponding to the root node, and the last one corresponding to the target node.

Graphical View of Interest Rate Tree

The function treeviewer provides a graphical view of the path of interest rates specified in BDTTree. For example, load the file deriv.mat. Here is a treeviewer representation of the rates along several branches of BDTTree.

```
    treeviewer(BDTTree)
```
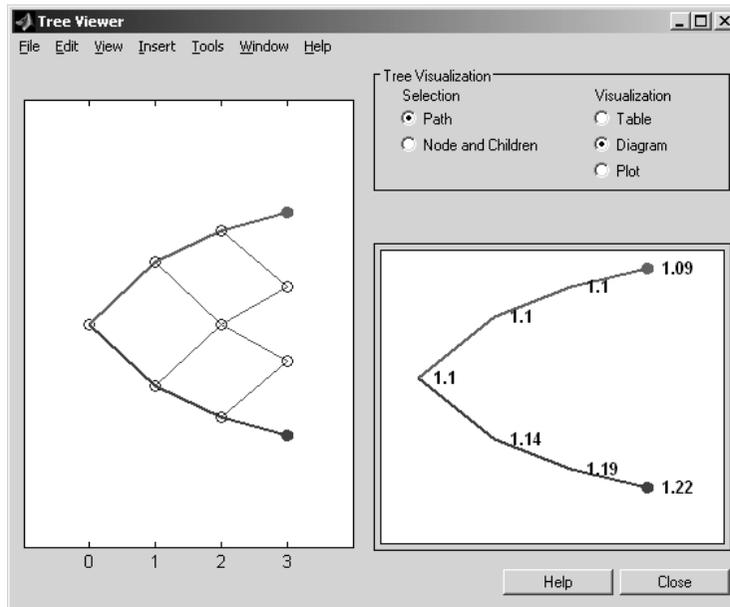
Note When using `treeviewer` with BDT trees, you must click on each node in succession from the beginning to the end. Because BDT trees can recombine, `treeviewer` is unable to compute the path automatically.

A previous example used `treepath` to find the path of interest rates taking the first branch up and then two branches down the rate tree.

```
FRates = treepath(BDTTree.FwdTree, [1 2 2])

FRates =

    1.1000
    1.0979
    1.1377
    1.1606
```

The `treeviewer` function displays the same information obtained by clicking along the sequence of nodes, as shown next.
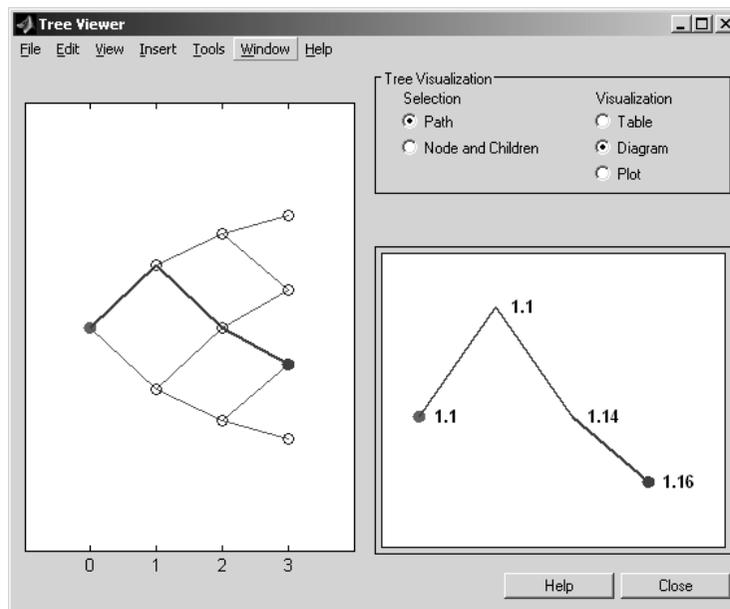
Pricing and Sensitivity from BDT

This section explains how to use the Financial Derivatives Toolbox to compute prices and sensitivities of several financial instruments using the Black-Derman-Toy (BDT) model. For information, see:

- "Pricing and the Price Tree" on page 2-63 for a discussion of using the bdtprice function to compute prices for a portfolio of instruments.
- "Calculating Prices and Sensitivities" on page 2-72 for a discussion of using the bdtsens function to compute delta, gamma, and vega portfolio sensitivities.

Pricing and the Price Tree

For the BDT model, the function bdtprice calculates the price of any set of supported instruments, based on an interest rate tree. The function is capable of pricing these instrument types:

- Bonds
- Bond options
- Arbitrary cash flows
- Fixed-rate notes
- Floating-rate notes
- Caps
- Floors
- Swaps

The syntax used for calling bdtprice is

```
[Price, PriceTree] = bdtprice(BDTTree, InstSet, Options)
```

This function requires two input arguments: the interest rate tree, BDTTree, and the set of instruments, InstSet. An optional argument Options further controls the pricing and the output displayed.

BDTTree is the Black-Derman-Toy tree sampling of an interest rate process, created using bdttree. See "Building a BDT Interest Rate Tree" on page 2-53 to learn how to create this structure based on the volatility model, the interest rate term structure, and the time layout.

InstSet is the set of instruments to be priced. This structure represents the set of instruments to be priced independently using the BDT model. The section "Creating and Managing Instrument Portfolios" on page 1-6 explains how to create this variable.

Options is an options structure created with the function derivset. This structure defines how the BDT tree is used to find the price of instruments in the portfolio, and how much additional information is displayed in the command window when calling the pricing function. If this input argument is not specified in the call to bdtprice, a default Options structure is used.

bdtprice classifies the instruments and calls appropriate pricing function for each of the instrument types. The pricing functions are bondbybdt, cfbybdt, fixedbybdt, floatbybdt, optbndbybdt, and swapbybdt. You can also use these functions directly to calculate the price of sets of instruments of the same type. See the documentation for these individual functions for further information.

BDT Pricing Example

Consider the following example, which uses the data in the MAT-file deriv.mat included in the toolbox. Load the data into the MATLAB workspace.

```
load deriv.mat
```

Use the MATLAB whos command to display a list of the variables loaded from the MAT-file.

```
whos

Name              Size            Bytes  Class

  BDTInstSet      1x1             22708  struct array
  BDTTree         1x1              5522  struct array
  HJMInstSet      1x1             22700  struct array
  HJMTree         1x1              6318  struct array
  ZeroInstSet     1x1             14442  struct array
  ZeroRateSpec    1x1              1580  struct array
```

BDTTree and BDTInstSet are the input arguments needed to call the function bdtprice.

Use the function instdisp to examine the set of instruments contained in the variable BDTInstSet.

```
                    instdisp(BDTInstSet)


Index Type CouponRate Settle      Maturity    Period Basis .........           Name      Quantity
1     Bond 0.1        01-Jan-2000 01-Jan-2003 1      NaN.........             10% bond   100
2     Bond 0.1        01-Jan-2000 01-Jan-2004 2      NaN.........             10% bond   50


Index Type    UnderInd OptSpec Strike ExerciseDates AmericanOpt Name      Quantity
3     OptBond 1        call    9501   Jan-2002      NaN         Option 95  -50


Index Type  CouponRate Settle      Maturity     FixedReset Basis Principal Name      Quantity
4     Fixed 0.10       01-Jan-2000 01-Jan-2003  1          NaN   NaN       10% Fixed 80


Index Type  Spread Settle      Maturity    FloatReset Basis Principal Name     Quantity
5     Float 20     01-Jan-2000 01-Jan-2003 1          NaN   NaN       20BP Float 8


Index Type Strike Settle      Maturity     CapReset Basis Principal Name    Quantity
6     Cap  0.15   01-Jan-2000 01-Jan-2004  1        NaN   NaN       15% Cap 30


Index Type  Strike Settle      Maturity     FloorReset Basis Principal Name      Quantity
7     Floor 0.09   01-Jan-2000 01-Jan-2004  1          NaN   NaN       9% Floor 40


Index Type LegRate     Settle      Maturity     LegReset Basis Principal LegType Name          Quantity
8     Swap [0.15 10] 01-Jan-2000 01-Jan-2003  [1  1]   NaN   NaN       [NaN]   15%/10BP Swap 10
```

Note that there are eight instruments in this portfolio set: two bonds, one bond option, one fixed rate note, one floating rate note, one cap, one floor, and one swap. Each instrument has a corresponding index that identifies the instrument prices in the price vector returned by bdtprice.

Now use bdtprice to calculate the price of each instrument in the instrument set.

```
[Price, PriceTree] = bdtprice(BDTTree, BDTInstSet)
Warning: Not all cash flows are aligned with the tree. Result will
be approximated.

Price =

    95.5030
    93.9079
     1.7657
    95.5030
   100.6054
     1.4863
```

```
    0.0245
    7.3032
```

Note The warning shown above appears because some of the cash flows for the second bond do not fall exactly on a tree node. This situation is discussed in "HJM Pricing Options Structure" on page 2-44.

Price Vector

The prices in the vector `Price` correspond to the prices at observation time zero (`tObs = 0`), which is defined as the valuation date of the interest rate tree. The instrument indexing within `Price` is the same as the indexing within `InstSet`. In this example, the prices in the `Price` vector correspond to the instruments in the following order.

```
InstNames = instget(BDTInstSet, 'FieldName','Name')

InstNames =

10% Bond
10% Bond
Option 95
10% Fixed
20BP Float
15% Cap
9% Floor
15%/10BP Swap
```

Consequently, in the `Price` vector, the fourth element, `95.5030`, represents the price of the fourth instrument (10% fixed-rate note); the sixth element, `1.4863`, represents the price of the sixth instrument (15% cap).

Price Tree Structure

The output price tree structure `PriceTree` holds all the pricing information. The first field of this structure, `FinObj`, indicates that this structure represents a price tree. The second field, `PTree` is the tree holding the price of the instruments in each node of the tree. The third field, `AITree` is the tree holding the accrued interest of the instruments in each node of the tree. The fourth

2-65

field, tObs, represents the observation time of each level of PTree and AITree, with units in terms of compounding periods.

The function treeviewer provides a graphical representation of the tree, allowing you to examine interactively the values on the nodes of the tree.

```
treeviewer(PriceTree, BDTInstSet)
```
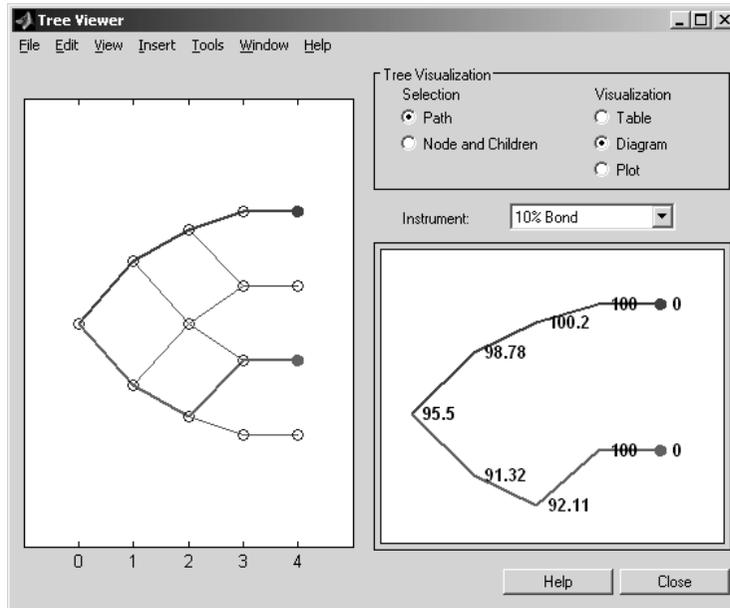

Alternatively, you can directly examine the field within the PriceTree structure, which contains the price tree with the price vectors at every state. The first node represents tObs = 0, corresponding to the valuation date.

```
PriceTree.PTree{1}

ans =

    95.5030
    93.9079
     1.7657
    95.5030
   100.6054
```

```
1.4863
0.0245
7.3032
```

You can also use `treeviewer` instrument-by-instrument to observe instrument prices. For the first 10% bond in the instrument portfolio, `treeviewer` indicates a valuation date price of `95.5030`, the same value obtained by accessing the `PriceTree` structure directly.
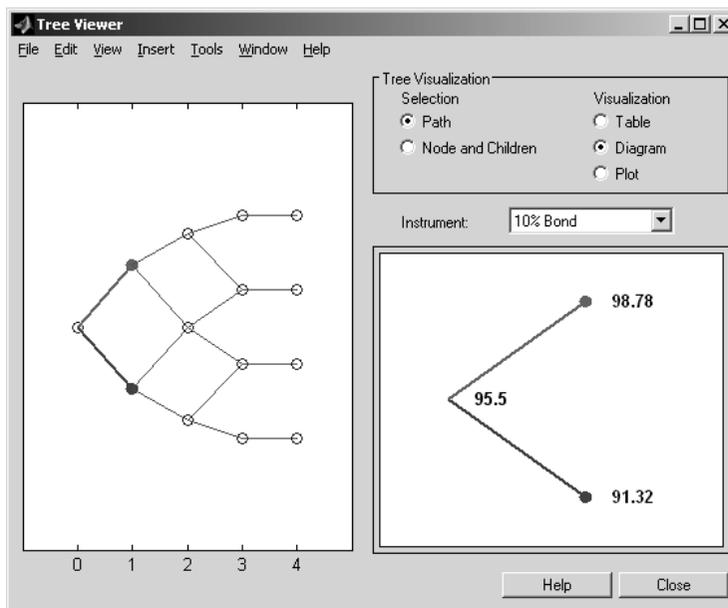
The second node represents the first rate observation time, `tObs = 1`. This node displays two states, one representing the branch going up and the other one representing the branch going down.

Examine the prices of the node corresponding to the up branch.

```
PriceTree.PTree{2}(:,1)

ans =

98.7816
97.9770
```

```
        3.1458
       98.7816
      101.9562
        0.5008
        0.0540
        5.6282
```

As before, you can use treeviewer, this time to examine the price for the 10% bond on the up branch. treeviewer displays a price of 98.7816 for the first node of the up branch, as expected.
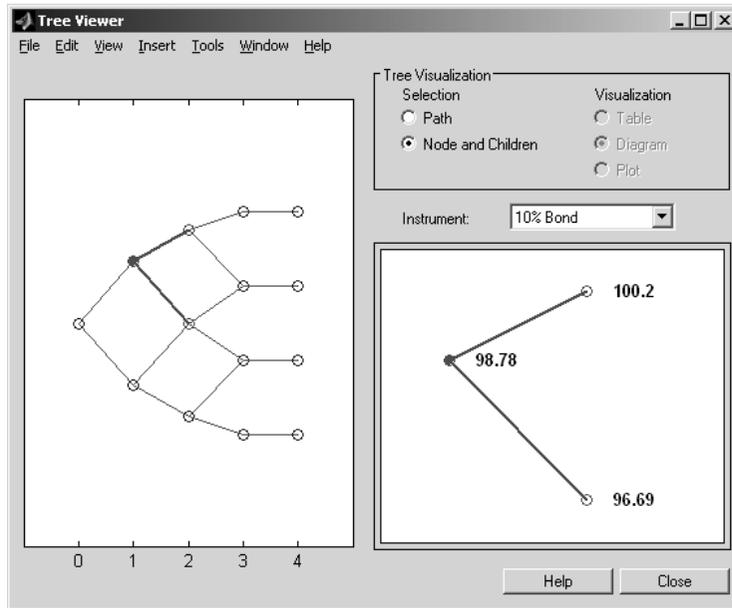
Now examine the corresponding down branch.

```
PriceTree.PTree{2}(:,2)

ans =

   91.3250
   88.1322
    0.7387
   91.3250
```

```
98.9758
 2.7691
      0
 0.6390
```

Use `treeviewer` once again, now to observe the price of the 10% bond on the down branch. The displayed price of `91.3250` conforms to the price obtained from direct access of the `PriceTree` structure. You may continue this process as far along the price tree as you want.
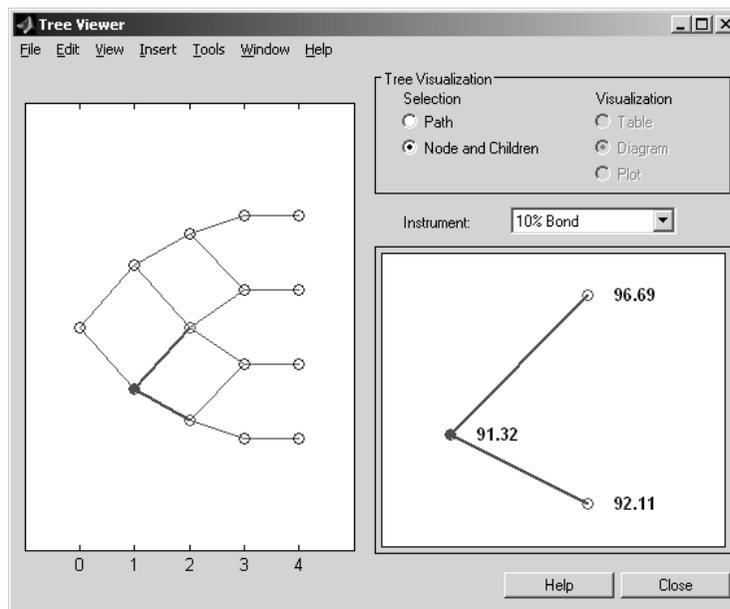
BDT Pricing Options Structure

The BDT instrument pricing functions (e.g., `bondbybdt`, `bdtprice`) use the same pricing options structure as their HJM counterparts. See the section "HJM Pricing Options Structure" on page 2-44 for a discussion of pricing options. You can obtain similar results from the examples in that section by substituting BDT functions and arguments for the corresponding HJM functions and arguments where required.

Calculating Prices and Sensitivities

The function `bdtsens` computes the delta, gamma, and vega sensitivities of instruments using an interest rate tree created with `bdttree`. It also optionally returns the calculated price for each instrument. `bdtsens` requires the same two input arguments used by `bdtprice`, namely `BDTTree` and `BDTInstSet`.

`bdtsens` calculates the dollar value of delta and gamma by shifting the observed forward yield curve by 100 basis points in each direction, and the dollar value of vega by shifting the volatility process by 1%. To obtain the per-dollar value of the sensitivities, divide the dollar sensitivity by the price of the corresponding instrument.

The calling syntax for the function is

```
[Delta, Gamma, Vega, Price] = bdtsens(BDTTree, BDTInstSet)
```

Use the previous example data to calculate the price of instruments.

```
load deriv.mat
[Delta, Gamma, Vega, Price] = bdtsens(BDTTree, BDTInstSet);
Warning: Not all cash flows are aligned with the tree. Result will
be approximated.
```

Note The warning appears because some of the cash flows for the second bond do not fall exactly on a tree node. This situation was discussed in "HJM Pricing Options Structure" on page 2-44.

You can conveniently examine the sensitivities and the prices by arranging them into a single matrix.

```
All = [Delta, Gamma, Vega, Price]
```

```
All =
1.0e+003 *

   -0.2327     0.8037    -0.0000     0.0955
   -0.2811     1.1819    -0.0000     0.0939
   -0.0505     0.2460     0.0053     0.0018
   -0.2327     0.8037          0     0.0955
    0.0000     0.0021     0.0014     0.1006
    0.0784     0.7490     0.0135     0.0015
   -0.0044     0.3821     0.0025     0.0000
   -0.2541     0.8642    -0.0014     0.0073
```

As with the prices, each row of the sensitivity vectors corresponds to the similarly indexed instrument in BDTInstSet. To view the *per-dollar sensitivities*, divide each dollar sensitivity by the corresponding instrument price.

```
All = [Delta ./ Price, Gamma ./ Price, Vega ./ Price, Price]

All =

  1.0e+004 *

   -0.0002     0.0008    -0.0000     0.0096
   -0.0003     0.0013    -0.0000     0.0094
   -0.0029     0.0139     0.0003     0.0002
   -0.0002     0.0008          0     0.0096
    0.0000     0.0000     0.0000     0.0101
    0.0053     0.0504     0.0009     0.0001
   -0.0178     1.5577     0.0102     0.0000
   -0.0035     0.0118    -0.0000     0.0007
```

3

Hedging Portfolios

Hedging

Hedging is an important consideration in modern finance. The decision of whether or not to hedge, how much portfolio insurance is adequate, and how often to rebalance a portfolio are important considerations for traders, portfolio managers, and financial institutions alike.

Without transaction costs, financial professionals would prefer to rebalance portfolios continually, thereby minimizing exposure to market movements. However, in practice, the transaction costs associated with frequent portfolio rebalancing may be very expensive. Therefore, traders and portfolio managers must carefully assess the cost needed to achieve a particular portfolio sensitivity (e.g., maintaining delta, gamma, and vega neutrality). Thus, the hedging problem involves the fundamental tradeoff between portfolio insurance and the cost of such insurance coverage.

The major topics covered in this chapter include:

- "Hedging Functions" on page 3-3
- "Self-Financing Hedges (hedgeslf)" on page 3-12
- "Specifying Constraints with ConSet" on page 3-16
- "Hedging with Constrained Portfolios" on page 3-21

Hedging Functions

The Financial Derivatives Toolbox offers two functions for assessing the fundamental hedging tradeoff, hedgeopt and hedgeslf.

The first function, hedgeopt, addresses the most general hedging problem. It allocates an optimal hedge to satisfy either of two goals:

- Minimize the cost of hedging a portfolio given a set of target sensitivities
- Minimize portfolio sensitivities for a given set of maximum target costs

hedgeopt allows investors to modify portfolio allocations among instruments according to either of the goals. The problem is cast as a constrained linear least squares problem. For additional information about hedgeopt, see "Hedging with hedgeopt" on page 3-3.

The second function, hedgeslf, attempts to allocate a self-financing hedge among a portfolio of instruments. In particular, hedgeslf attempts to maintain a constant portfolio value consistent with reduced portfolio sensitivities (i.e., the rebalanced portfolio is hedged against market moves and is closest to being self-financing). If hedgeslf cannot find a self-financing hedge, it rebalances the portfolio to minimize overall portfolio sensitivities. For additional information on hedgeslf, see "Self-Financing Hedges (hedgeslf)" on page 3-12.

Hedging with hedgeopt

To illustrate the hedging functions, consider the *delta*, *gamma*, and *vega* sensitivity measures. In the context of the Financial Derivatives Toolbox, delta is the price sensitivity measure of shifts in the forward yield curve, gamma is the delta sensitivity measure of shifts in the forward yield curve, and vega is the price sensitivity measure of shifts in the volatility process. Note that the delta, gamma, and vega sensitivities calculated by the toolbox are dollar sensitivities. (See "Calculating Prices and Sensitivities" on page 2-50 (HJM) and "Calculating Prices and Sensitivities" on page 2-72 (BDT) for details.)

Note The numerical results in this section are displayed with the MATLAB bank format. Although the calculations are performed in floating-point double precision, only two decimal places are displayed.

To illustrate the hedging facility, consider the portfolio `HJMInstSet` obtained from the example file `deriv.mat`. The portfolio consists of eight instruments: two bonds, one bond option, one fixed rate note, one floating rate note, one cap, one floor, and one swap.

Both hedging functions require some common inputs, including the current portfolio holdings (allocations), and a matrix of instrument sensitivities. To create these inputs, load the example portfolio into memory

```
load deriv.mat;
```

compute price and sensitivities

```
[Delta, Gamma, Vega, Price] = hjmsens(HJMTree, HJMInstSet);
Warning: Not all cash flows are aligned with the tree. Result will
be approximated.
```

and extract the current portfolio holdings.

```
Holdings = instget(HJMInstSet, 'FieldName', 'Quantity');
```

For convenience place the delta, gamma, and vega sensitivity measures into a matrix of sensitivities.

```
Sensitivities = [Delta Gamma Vega];
```

Each row of the `Sensitivities` matrix is associated with a different instrument in the portfolio, and each column with a different sensitivity measure.

To summarize the portfolio information

```
disp([Price  Holdings  Sensitivities])

  98.72       100.00      -272.65       1029.90        0.00
  97.53        50.00      -347.43       1622.69       -0.04
   0.05       -50.00        -8.08        643.40       34.07
  98.72        80.00      -272.65       1029.90        0.00
 100.55         8.00        -1.04          3.31           0
   6.28        30.00       294.97       6852.56       93.69
   0.05        40.00       -47.16       8459.99       93.69
   3.69        10.00      -282.05       1059.68        0.00
```

The first column above is the dollar unit price of each instrument, the second is the holdings of each instrument (the quantity held or the number of

contracts), and the third, fourth, and fifth columns are the dollar delta, gamma, and vega sensitivities, respectively.

The current portfolio sensitivities are a weighted average of the instruments in the portfolio.

```
TargetSens  = Holdings' * Sensitivities

TargetSens =

    -61910.22     788946.21        4852.91
```

Maintaining Existing Allocations

To illustrate using hedgeopt, suppose that you want to maintain your existing portfolio. The first form of hedgeopt minimizes the cost of hedging a portfolio given a set of target sensitivities. If you want to maintain your existing portfolio composition and exposure, you should be able to do so without spending any money. To verify this, set the target sensitivities to the current sensitivities.

```
[Sens, Cost, Quantity] = hedgeopt(Sensitivities, Price,...
Holdings, [], [], [], TargetSens)

Sens =

    -61910.22     788946.21        4852.91

Cost =

            0

Quantity' =

        100.00
         50.00
        -50.00
         80.00
          8.00
         30.00
         40.00
         10.00
```

Our portfolio composition and sensitivities are unchanged, and the cost associated with doing nothing is zero. The cost is defined as the change in portfolio value. This number cannot be less than zero because the rebalancing cost is defined as a nonnegative number.

If `Value0` and `Value1` represent the portfolio value before and after rebalancing, respectively, the zero cost can also be verified by comparing the portfolio values.

```
Value0 = Holdings' * Price

Value0 =

    23674.62

Value1 = Quantity * Price

Value1 =

    23674.62
```

Partially Hedged Portfolio

Building upon the previous example, suppose you want to know the cost to achieve an overall portfolio dollar sensitivity of [-23000 -3300 3000], while allowing trading only in instruments 2, 3, and 6 (holding the positions of instruments 1, 4, 5, 7, and 8 fixed.) To find the cost, first set the target portfolio dollar sensitivity.

```
TargetSens = [-23000 -3300 3000];
```

Then, specify the instruments to be fixed.

```
FixedInd = [1 4 5 7 8];
```

Finally, call `hedgeopt`

```
[Sens, Cost, Quantity] = hedgeopt(Sensitivities, Price,...
Holdings, FixedInd, [], [], TargetSens);
```

and again examine the results.

```
Sens =
```

```
        -23000.00        -3300.00         3000.00

    Cost =

          19174.02

    Quantity' =

              100.00
             -141.03
              137.26
               80.00
                8.00
              -57.96
               40.00
               10.00
```

Recompute Value1, the portfolio value after rebalancing.

```
    Value1 = Quantity * Price

    Value1 =

          4500.60
```

As expected, the cost, $19174.02, is the difference between Value0 and Value1, $23674.62 - $4500.60. Only the positions in instruments 2, 3, and 6 have been changed.

Fully Hedged Portfolio

The above example illustrates a partial hedge, but perhaps the most interesting case involves the cost associated with a fully-hedged portfolio (simultaneous delta, gamma, and vega neutrality). In this case, set the target sensitivity to a row vector of zeros and call hedgeopt again.

```
    TargetSens = [0 0 0];
    [Sens, Cost, Quantity] = hedgeopt(Sensitivities, Price, ...
    Holdings, FixedInd, [], [], TargetSens);
```

Examining the outputs reveals that you have obtained a fully-hedged portfolio

```
    Sens =
```

```
          -0.00              -0.00              -0.00
```

but at an expense of over $20,000,

```
  Cost =

       23055.90
```

The positions needed to achieve a fully-hedged portfolio

```
  Quantity' =

           100.00
          -182.36
           -19.55
            80.00
             8.00
           -32.97
            40.00
            10.00
```

result in the new portfolio value

```
  Value1 = Quantity * Price

  Value1 =

        618.72
```

Minimizing Portfolio Sensitivities

The above examples illustrate how to use `hedgeopt` to determine the minimum cost of hedging a portfolio given a set of target sensitivities. In these examples, portfolio target sensitivities are treated as equality constraints during the optimization process. You tell `hedgeopt` what sensitivities you want, and it tells you what it will cost to get those sensitivities.

A related problem involves minimizing portfolio sensitivities for a given set of maximum target costs. For this goal the target costs are treated as inequality constraints during the optimization process. You tell `hedgeopt` the most you are willing spend to insulate your portfolio, and it tells you the smallest portfolio sensitivities you can get for your money.

To illustrate this use of hedgeopt, compute the portfolio dollar sensitivities along the entire cost frontier. From the previous examples, you know that spending nothing simply replicates the existing portfolio, while spending $23,055.90 completely hedges the portfolio.

Assume, for example, you are willing to spend as much as $50,000, and want to see what portfolio sensitivities will result along the cost frontier. Assume the same instruments are held fixed, and that the cost frontier is evaluated from $0 to $50,000 at increments of $1000.

```
MaxCost = [0:1000:50000];
```

Now, call hedgeopt.

```
[Sens, Cost, Quantity] = hedgeopt(Sensitivities, Price, ...
Holdings, FixedInd, [], MaxCost);
```

With this data, you can plot the required hedging cost versus the funds available (the amount you are willing to spend).

```
plot(MaxCost/1000, Cost/1000, 'red'), grid
xlabel('Funds Available for Rebalancing ($1000''s)')
ylabel('Actual Rebalancing Cost ($1000''s)')
title ('Rebalancing Cost Profile')
```
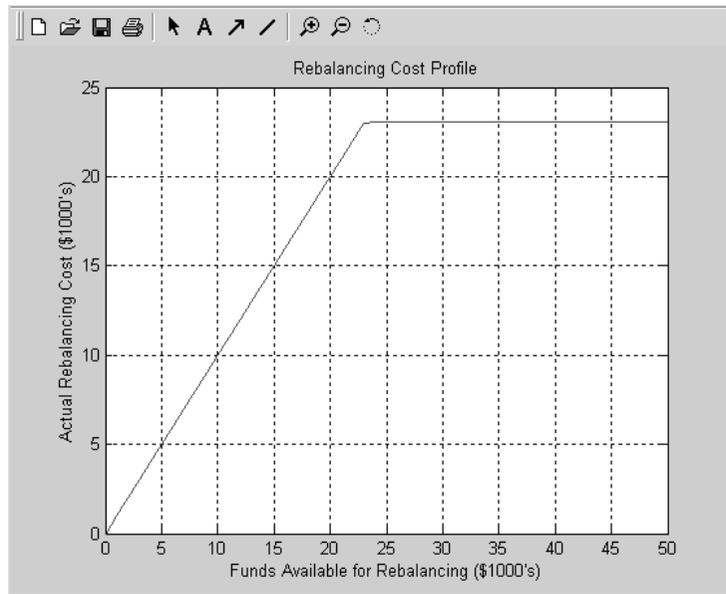
Figure 3-1: Rebalancing Cost Profile

and the portfolio dollar sensitivities versus the funds available

```
figure
plot(MaxCost/1000, Sens(:,1), '-red')
hold('on')
plot(MaxCost/1000, Sens(:,2), '-.black')
plot(MaxCost/1000, Sens(:,3), '--blue')
grid
xlabel('Funds Available for Rebalancing ($1000''s)')
ylabel('Delta, Gamma, and Vega Portfolio Dollar Sensitivities')
title ('Portfolio Sensitivities Profile')
legend('Delta', 'Gamma', 'Vega', 0)
```
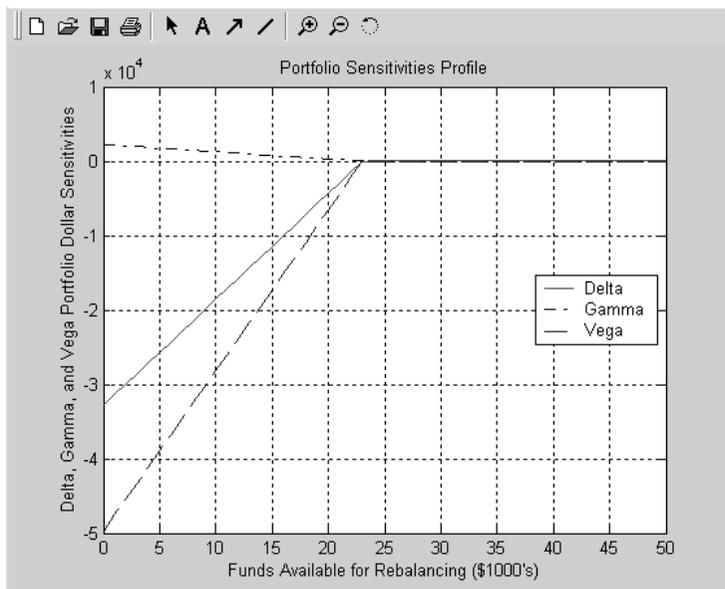
Figure 3-2: Funds Available for Rebalancing

Self-Financing Hedges (hedgeslf)

Figure 3-1 and Figure 3-2 indicate that there is no benefit to be gained because the funds available for hedging exceed $23,055.90, the point of maximum expense required to obtain simultaneous delta, gamma, and vega neutrality. You can also find this point of delta, gamma, and vega neutrality using `hedgeslf`.

```
[Sens, Value1, Quantity] = hedgeslf(Sensitivities, Price,...
Holdings, FixedInd);

Sens =

        -0.00
        -0.00
        -0.00

Value1 =

       618.72

Quantity =

        100.00
       -182.36
        -19.55
         80.00
          8.00
        -32.97
         40.00
         10.00
```

Similar to `hedgeopt`, `hedgeslf` returns the portfolio dollar sensitivities and instrument quantities (the rebalanced holdings). However, in contrast, the second output parameter of `hedgeslf` is the value of the rebalanced portfolio, from which you can calculate the rebalancing cost by subtraction.

```
Value0 - Value1

ans =
```

```
        23055.90
```

In our example, the portfolio is clearly not self-financing, so hedgeslf finds the best possible solution required to obtain zero sensitivities.

There is, in fact, a third calling syntax available for hedgeopt directly related to the results shown above for hedgeslf. Suppose, instead of directly specifying the funds available for rebalancing (the most money you are willing to spend), you want to simply specify the number of points along the cost frontier. This call to hedgeopt samples the cost frontier at 10 equally spaced points between the point of minimum cost (and potentially maximum exposure) and the point of minimum exposure (and maximum cost).

```
[Sens, Cost, Quantity] = hedgeopt(Sensitivities, Price,...
Holdings, FixedInd, 10);

Sens =
    -32784.46        2231.83      -49694.33
    -29141.74        1983.85      -44172.74
    -25499.02        1735.87      -38651.14
    -21856.30        1487.89      -33129.55
    -18213.59        1239.91      -27607.96
    -14570.87         991.93      -22086.37
    -10928.15         743.94      -16564.78
     -7285.43         495.96      -11043.18
     -3642.72         247.98       -5521.59
         0.00          -0.00          0.00

Cost =
          0.00
       2561.77
       5123.53
       7685.30
      10247.07
      12808.83
      15370.60
      17932.37
      20494.14
      23055.90
```

Now plot this data.

```
figure
plot(Cost/1000, Sens(:,1), '-red')
hold('on')
plot(Cost/1000, Sens(:,2), '-.black')
plot(Cost/1000, Sens(:,3), '--blue')
grid
xlabel('Rebalancing Cost ($1000''s)')
ylabel('Delta, Gamma, and Vega Portfolio Dollar Sensitivities')
title ('Portfolio Sensitivities Profile')
legend('Delta', 'Gamma', 'Vega', 0)
```
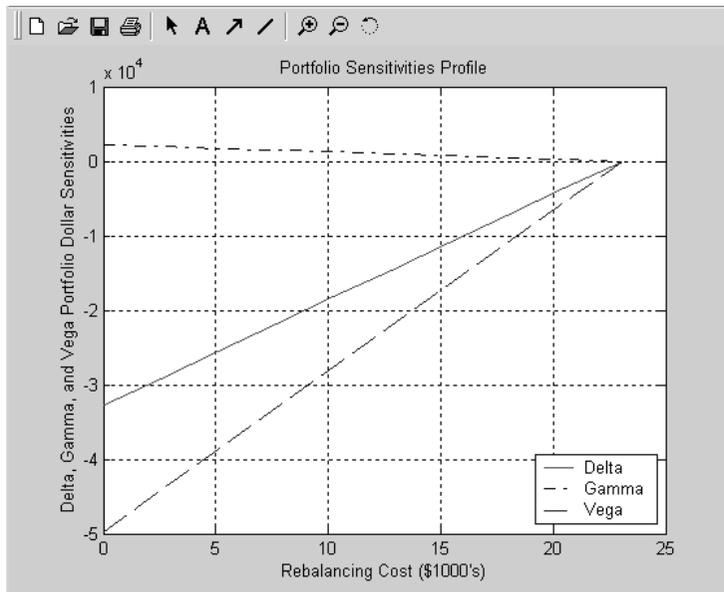

Figure 3-3: Rebalancing Cost

In this calling form, hedgeopt calls hedgeslf internally to determine the maximum cost needed to minimize the portfolio sensitivities ($23,055.90), and evenly samples the cost frontier between $0 and $23,055.90.

Note that both hedgeopt and hedgeslf cast the optimization problem as a constrained linear least squares problem. Depending upon the instruments and constraints, neither function is guaranteed to converge to a solution. In some cases, the problem space may be unbounded, and additional instrument

equality constraints, or user-specified constraints, may be necessary for convergence. See "Hedging with Constrained Portfolios" on page 3-21 for additional information.

Specifying Constraints with ConSet

Both `hedgeopt` and `hedgeslf` accept an optional input argument, `ConSet`, that allows you to specify a set of linear inequality constraints for instruments in your portfolio. The examples in this section are quite brief. For additional information regarding portfolio constraint specifications, refer to the section "Analyzing Portfolios" found in the *Financial Toolbox User's Guide*.

Setting Constraints

For the first example of setting constraints, return to the fully-hedged portfolio example that used `hedgeopt` to determine the minimum cost of obtaining simultaneous delta, gamma, and vega neutrality (target sensitivities all zero). Recall that when `hedgeopt` computes the cost of rebalancing a portfolio, the input target sensitivities you specify are treated as equality constraints during the optimization process. The situation is reproduced below for convenience.

```
TargetSens = [0 0 0];
[Sens, Cost, Quantity] = hedgeopt(Sensitivities, Price,...
Holdings, FixedInd, [], [], TargetSens);
```

The outputs provide a fully-hedged portfolio

```
Sens =
        -0.00          -0.00          -0.00
```

at an expense of over $23,000.

```
Cost =
     23055.90
```

The positions needed to achieve this fully-hedged portfolio are

```
Quantity' =

         100.00
        -182.36
         -19.55
          80.00
           8.00
         -32.97
          40.00
          10.00
```

Suppose now that you want to place some upper and lower bounds on the individual instruments in your portfolio. You can specify these constraints, along with a variety of general linear inequality constraints, with the Financial Toolbox function portcons.

As an example, assume that, in addition to holding instruments 1, 4, 5, 7, and 8 fixed as before, you want to bound the position of all instruments to within +/- 180 contracts (for each instrument, you cannot short or long more than 180 contracts). Applying these constraints disallows the current position in the second instrument (short 182.36). All other instruments are currently within the upper/lower bounds.

You can generate these constraints by first specifying the lower and upper bounds vectors and then calling portcons.

```
LowerBounds = [-180 -180 -180 -180 -180 -180 -180 -180];
UpperBounds = [ 180  180   180 180  180  180  180  180];
ConSet = portcons('AssetLims', LowerBounds, UpperBounds);
```

To impose these constraints, call hedgeopt with ConSet as the last input.

```
[Sens, Cost, Quantity] = hedgeopt(Sensitivities, Price,...
Holdings, FixedInd, [], [], TargetSens, ConSet);
```

Examine the outputs and see that they are all set to NaN, indicating that the problem, given the constraints, is not solvable. Intuitively, the results mean that you cannot obtain simultaneous delta, gamma, and vega neutrality with these constraints at any price.

To see how close you can get to portfolio neutrality with these constraints, call hedgeslf.

```
[Sens, Value1, Quantity] = hedgeslf(Sensitivities, Price,...
Holdings, FixedInd, ConSet);

Sens =

      -352.43
        21.99
      -498.77

Value1 =
```

```
          855.10

Quantity =

          100.00
         -180.00
          -37.22
           80.00
            8.00
          -31.86
           40.00
           10.00
```

hedgeslf enforces the lower bound for the second instrument, but the
sensitivity is far from neutral. The cost to obtain this portfolio is

```
Value0 - Value1

ans =

       22819.52
```

Portfolio Rebalancing

As a final example of user-specified constraints, rebalance the portfolio using
the second hedging goal of hedgeopt. Assume that you are willing to spend as
much as $20,000 to rebalance your portfolio, and you want to know what
minimum portfolio sensitivities you can get for your money. In this form, recall
that the target cost ($20,000) is treated as an inequality constraint during the
optimization process.

For reference, invoke hedgeopt without any user-specified linear inequality
constraints.

```
[Sens, Cost, Quantity] = hedgeopt(Sensitivities, Price,...
Holdings, FixedInd, [], 20000);

Sens =

    -4345.36         295.81       -6586.64
Cost =
```

```
       20000.00

Quantity' =

         100.00
        -151.86
        -253.47
          80.00
           8.00
         -18.18
          40.00
          10.00
```

This result corresponds to the $20,000 point along the Portfolio Sensitivities Profile shown in Figure 3-3, Rebalancing Cost, on page 3-14.

Assume that, in addition to holding instruments 1, 4, 5, 7, and 8 fixed as before, you want to bound the position of all instruments to within +/- 150 contracts (for each instrument, you cannot short more than 150 contracts and you cannot long more than 150 contracts). These bounds disallow the current position in the second and third instruments (-151.86 and -253.47). All other instruments are currently within the upper/lower bounds.

As before, you can generate these constraints by first specifying the lower and upper bounds vectors and then calling portcons.

```
LowerBounds = [-150 -150 -150 -150 -150 -150 -150 -150];
UpperBounds = [ 150  150  150  150  150  150  150  150];
ConSet = portcons('AssetLims', LowerBounds, UpperBounds);
```

To impose these constraints, again call hedgeopt with ConSet as the last input.

```
[Sens, Cost, Quantity] = hedgeopt(Sensitivities, Price,...
Holdings,FixedInd, [], 20000, [], ConSet);

Sens =

      -8818.47         434.43        -4010.79

Cost =

      19876.89
```

```
Quantity' =

        100.00
       -150.00
       -150.00
         80.00
          8.00
        -28.32
         40.00
         10.00
```

With these constraints hedgeopt enforces the lower bound for the second and third instruments. The cost incurred is $19,876.89.

Hedging with Constrained Portfolios

Both hedging functions cast the optimization as a constrained linear least squares problem. (See the function lsqlin in the Optimization Toolbox for details.) In particular, lsqlin attempts to minimize the constrained linear least squares problem

$$\min_{x} \frac{1}{2} \|Cx - d\|_2^2 \qquad \text{such that} \qquad \begin{aligned} A \cdot x &\le b \\ Aeq \cdot x &= beq \\ lb &\le x \le ub \end{aligned}$$

where *C*, *A*, and *Aeq* are matrices, and *d*, *b*, *beq*, *lb*, and *ub* are vectors. In all cases of interest for the Financial Derivatives Toolbox, *x* is a vector of asset holdings (contracts).

This section provides some examples of setting constraints and discusses how to recognize situations when the least squares problem is improperly constrained. Depending upon the constraints and the number of assets in the portfolio, a solution to a particular problem may or may not exist. Furthermore, if a solution is found, the solution may not be unique. For a unique solution to exist, the least squares problem must be sufficiently and appropriately constrained.

Example: Fully Hedged Portfolio

Recall that hedgeopt allows you to allocate an optimal hedge by one of two goals:

- Minimize the cost of hedging a portfolio given a set of target sensitivities
- Minimize portfolio sensitivities for a given set of maximum target costs

As an example, reproduce the results for the fully hedged portfolio example.

```
TargetSens = [0 0 0];
FixedInd  = [1 4 5 7 8];
[Sens,Cost,Quantity] = hedgeopt(Sensitivities, Price,...
Holdings, FixedInd, [], [], TargetSens);


Sens =
```

```
              -0.00              -0.00              -0.00

  Cost =

        23055.90

  Quantity' =

            98.72
          -182.36
           -19.55
            80.00
             8.00
           -32.97
            40.00
            10.00
```

This example finds a unique solution at a cost of just over $23,000. The matrix C (formed internally by hedgeopt and passed to lsqlin) is the asset Price vector expressed as a row vector.

```
  C = Price' = [98.72 97.53 0.05 98.72 100.55 6.28 0.05 3.69]
```

The vector d is the current portfolio value Value0 = 23674.62. The example maintains, as closely as possible, a constant portfolio value subject to the specified constraints.

Additional Constraints

In the absence of any additional constraints, the least squares objective involves a single equation with eight unknowns. This is an under-determined system of equations. Because such systems generally have an infinite number of solutions, you need to specify additional constraints to achieve a solution with practical significance. The additional constraints can come from two sources:

- User-specified equality constraints
- Target sensitivity equality constraints imposed by hedgeopt

The fully-hedged portfolio example specifies five equality constraints associated with holding assets 1, 4, 5, 7, and 8 fixed. This reduces the number

of unknowns from eight to three, which is still an under-determined system. However, when combined with the first goal of hedgeopt, the equality constraints associated with the target sensitivities in TargetSens produce an additional system of three equations with three unknowns. This additional system guarantees that the weighted average of the delta, gamma, and vega of assets 2, 3, and 6, together with the remaining assets held fixed, satisfy the overall portfolio target sensitivity requirements in TargetSens.

Combining the least squares objective equation with the three portfolio sensitivity equations provides an overall system of four equations with three unknown asset holdings. This is no longer an under-determined system, and the solution is as shown.

If the assets held fixed are reduced, e.g., FixedInd = [1 4 5 7], hedgeopt returns a no cost, fully-hedged portfolio (Sens = [0 0 0] and Cost = 0).

If you further reduce FixedInd (e.g., [1 4 5], [1 4], or even []), hedgeopt always returns a no cost, fully-hedged portfolio. In these cases, insufficient constraints result in an under-determined system. Although hedgeopt identifies no cost, fully-hedged portfolios, there is nothing unique about them. These portfolios have little practical significance.

Constraints must be *sufficient* and *appropriately defined*. Additional constraints having no effect on the optimization are called *dependent constraints*. As a simple example, assume that parameter Z is constrained such that $Z \leq 1$. Furthermore, assume we somehow add another constraint that effectively restricts $Z \leq 0$. The constraint $Z \leq 1$ now has no effect on the optimization.

Example: Minimize Portfolio Sensitivities

To illustrate using hedgeopt to minimize portfolio sensitivities for a given maximum target cost, specify a target cost of $20,000 and determine the new portfolio sensitivities, holdings, and cost of the rebalanced portfolio.

```
MaxCost = 20000;
[Sens, Cost, Quantity] = hedgeopt(Sensitivities, Price,...
Holdings, [1 4 5 7 8], [], MaxCost);

Sens =

    -4345.36        295.81       -6586.64
```

```
Cost =

        20000.00

Quantity' =

         100.00
        -151.86
        -253.47
          80.00
           8.00
         -18.18
          40.00
          10.00
```

This example corresponds to the $20,000 point along the cost axis in Figure 3-1, Figure 3-2, and Figure 3-3.

When minimizing sensitivities, the maximum target cost is treated as an inequality constraint; in this case, `MaxCost` is the most you are willing to spend to hedge a portfolio. The least squares objective matrix `C` is the matrix transpose of the input asset sensitivities

```
C = Sensitivities'
```

a 3-by-8 matrix in this example, and d is a 3-by-1 column vector of zeros, [0 0 0]'.

Without any additional constraints, the least squares objective results in an under-determined system of three equations with eight unknowns. By holding assets 1, 4, 5, 7, and 8 fixed, you reduce the number of unknowns from eight to three. Now, with a system of three equations with three unknowns, `hedgeopt` finds the solution shown.

Example: Under-Determined System

Reducing the number of assets held fixed creates an under-determined system with meaningless solutions. For example, see what happens with only four assets constrained.

```
FixedInd = [1 4 5 7];
```

```
[Sens, Cost, Quantity] = hedgeopt(Sensitivities, Price,...
Holdings, FixedInd, [], MaxCost);

Sens =

        -0.00           -0.00           -0.00

Cost =

      20000.00

Quantity' =

        100.00
       -149.31
        -14.91
         80.00
          8.00
        -34.64
         40.00
        -32.60
```

You have spent $20,000 (all the funds available for rebalancing) to achieve a fully-hedged portfolio.

With an increase in available funds to $50,000, you still spend all available funds to get another fully-hedged portfolio.

```
MaxCost  = 50000;
[Sens, Cost, Quantity] = hedgeopt(Sensitivities, Price,...
Holdings, FixedInd, [],MaxCost);

Sens =

        -0.00            0.00            0.00
Cost =

      50000.00

Quantity' =
```

```
        100.00
       -473.78
        -60.51
         80.00
          8.00
        -18.20
         40.00
        385.60
```

All solutions to an under-determined system are meaningless. You buy and sell various assets to obtain zero sensitivities, spending all available funds every time. If you reduce the number of fixed assets any further, this problem is insufficiently constrained, and you find no solution (the outputs are all `NaN`).

Note also that no solution exists whenever constraints are *inconsistent*. Inconsistent constraints create an infeasible solution space; the outputs are all `NaN`.

Portfolio Constraints with hedgeslf

The other hedging function, `hedgeslf`, attempts to minimize portfolio sensitivities such that the rebalanced portfolio maintains a constant value (the rebalanced portfolio is hedged against market moves and is closest to being self-financing). If a self-financing hedge is not found, `hedgeslf` tries to rebalance a portfolio to minimize sensitivities.

From a least squares systems approach, `hedgeslf` first attempts to minimize cost in the same way that `hedgeopt` does. If it cannot solve this problem (a no cost, self-financing hedge is not possible), `hedgeslf` proceeds to minimize sensitivities like `hedgeopt`. Thus, the discussion of constraints for `hedgeopt` is directly applicable to `hedgeslf` as well.

Function Reference

Functions - By Category

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

- "Portfolio Hedge Allocation"
- "Fixed Income Pricing from Interest Term Structure"
- "Fixed Income Pricing and Sensitivity from Heath-Jarrow-Morton Tree"
- "Fixed Income Pricing and Sensitivity from Black-Derman-Toy Tree"
- "Heath-Jarrow-Morton Utilities"
- "Black-Derman-Toy Utilities"
- "Heath-Jarrow-Morton Bushy Tree Manipulation"
- "Black-Derman-Toy Recombining Tree Manipulation"
- "Derivatives Pricing Options"
- "Instrument Portfolio Handling"
- "Financial Object Structures"
- "Interest Term Structure"
- "Date Functions"
- "Graphical Display Functions"

Portfolio Hedge Allocation

hedgeslf   Self-financing hedge

hedgeopt   Allocate optimal hedge for target costs or sensitivities

Fixed Income Pricing from Interest Term Structure

bondbyzero   Price bond by a set of zero curves

cfbyzero   Price cash flows by a set of zero curves

fixedbyzero   Price fixed rate note by a set of zero curves

floatbyzero   Price floating rate note by a set of zero curves

intenvprice   Price fixed income instruments by a set of zero curves

intenvsens   Instrument prices and sensitivities by a set of zero curves

swapbyzero   Price swap instrument by a set of zero curves

Fixed Income Pricing and Sensitivity from Heath-Jarrow-Morton Tree

hjmprice   Fixed income instrument prices by HJM interest rate tree

hjmsens   Fixed income instrument prices and sensitivities by HJM interest rate tree

hjmtimespec   Specify time structure for HJM interest rate tree

hjmtree   Construct HJM interest rate tree

hjmvolspec   HJM volatility process specification

Fixed Income Pricing and Sensitivity from Black-Derman-Toy Tree

bdtprice   Fixed income instrument prices by BDT interest rate tree

bdtsens   Fixed income instrument prices and sensitivities by BDT interest rate tree

bdttimespec   Specify time structure for BDT interest rate tree

bdttree   Construct BDT interest rate tree

bdtvolspec   BDT volatility process specification

Heath-Jarrow-Morton Utilities

bondbyhjm	Price bond by HJM interest rate tree
capbyhjm	Price cap instrument by HJM interest rate tree
cfbyhjm	Price arbitrary set of cash flows by HJM interest rate tree
fixedbyhjm	Price fixed rate note by HJM interest rate tree
floatbyhjm	Price floating rate note by HJM interest rate tree
floorbyhjm	Price floor instrument by HJM interest rate tree
mmktbyhjm	Create money market tree from HJM
optbndbyhjm	Price bond option by HJM interest rate tree
swapbyhjm	Price swap instrument by HJM interest rate tree

Black-Derman-Toy Utilities

bondbybdt	Price bond by BDT interest rate tree
capbybdt	Price cap by BDT interest rate tree
cfbybdt	Price arbitrary set of cash flows by BDT interest rate tree
fixedbybdt	Price fixed rate note by BDT interest rate tree
floatbybdt	Price floating rate note by BDT interest rate tree
floorbybdt	Price floor instrument by BDT interest rate tree
mmktbybdt	Create money market tree from BDT
optbndbybdt	Price bond option by BDT interest rate tree
swapbybdt	Price swap instrument by BDT interest rate tree

Heath-Jarrow-Morton Bushy Tree Manipulation

bushpath	Extract entries from node of bushy tree
bushshape	Retrieve shape of bushy tree
mkbush	Create bushy tree

Black-Derman-Toy Recombining Tree Manipulation

mktree	Create recombining tree
treepath	Extract entries from node of recombining tree
treeshape	Retrieve shape of recombining tree

Derivatives Pricing Options

derivget	Get derivatives pricing options
derivset	Set or modify derivatives pricing options

Instrument Portfolio Handling

instadd	Add types to instrument collection
instaddfield	Add new instruments to an instrument collection
instbond	Construct bond instrument
instcap	Construct cap instrument
instcf	Constructor for arbitrary cash flow instrument
instdelete	Complement of subset of instruments by matching conditions
instdisp	Display instruments
instfields	List fieldnames
instfind	Search instruments for matching conditions
instfixed	Construct fixed-rate instrument
instfloat	Construct floating-rate instrument
instfloor	Construct floor instrument
instget	Retrieve data from instrument variable
instgetcell	Retrieve data and context from instrument variable
instlength	Count instruments
instoptbnd	Construct bond option

instselect	Create instrument subset by matching conditions
instsetfield	Add or reset data for existing instruments
instswap	Construct swap instrument
insttypes	List types

Financial Object Structures

| classfin | Create financial structure or return financial structure class name |
| isafin | True if financial structure type or financial object class |

Interest Term Structure

date2time	Fixed income time and frequency from dates
disc2rate	Interest rates from cash flow discounting factors
intenvget	Get properties of interest rate environment
intenvset	Set properties of interest rate environment
rate2disc	Discounting factors from interest rates
ratetimes	Change time intervals defining interest rate environment

Date Functions

| datedisp | Display date entries |

Graphical Display Functions

| treeviewer | Display tree information |

4

Functions - Alphabetical List

4

bdtprice

Purpose	Fixed income instrument prices by BDT interest rate tree

Syntax   [Price, PriceTree] = bdtprice(BDTTree, InstSet, Options)

Arguments

BDTTree	Interest rate tree structure created by bdttree.
InstSet	Variable containing a collection of NINST instruments. Instruments are categorized by type; each type can have different data fields. The stored data field is a row vector or string for each instrument.
Options	(Optional) Derivatives pricing options structure created with derivset.

Description   [Price, PriceTree] = bdtprice(BDTTree, InstSet, Options) computes arbitrage free prices for instruments using an interest rate tree created with bdttree. All instruments contained in a financial instrument variable, InstSet, are priced.

Price is a number of instruments (NINST)-by-1 vector of prices for each instrument. The prices are computed by backward dynamic programming on the interest rate tree. If an instrument cannot be priced, NaN is returned.

PriceTree is a MATLAB structure of trees containing vectors of instrument prices and accrued interest, and a vector of observation times for each node.

PriceTree.PTree contains the clean prices.

PriceTree.AITree contains the accrued interest.

PriceTree.tObs contains the observation times.

bdtprice handles instrument types: 'Bond', 'CashFlow', 'OptBond', 'Fixed', 'Float', 'Cap', 'Floor', 'Swap'. See instadd to construct defined types.

Related single-type pricing functions are:

- bondbybdt: Price a bond by a BDT tree.
- capbybdt: Price a cap by a BDT tree.
- cfbybdt: Price an arbitrary set of cash flows by a BDT tree.
- fixedbybdt: Price a fixed rate note by a BDT tree.
- floatbybdt: Price a floating rate note by a BDT tree.

bdtprice

- `floorbybdt`: Price a floor by a BDT tree.
- `optbndbybdt`: Price a bond option by a BDT tree.
- `swapbybdt`: Price a swap by a BDT tree.

Examples   Load the BDT tree and instruments from the data file `deriv.mat`. Price the cap and bond instruments contained in the instrument set.

```
load deriv.mat;
BDTSubSet = instselect(BDTInstSet,'Type', {'Bond', 'Cap'});

instdisp(BDTSubSet)
```

```
Index Type   CouponRate Settle       Maturity    Period  Name ...
1     Bond   0.1        01-Jan-2000 01-Jan-2003 1        10% bond
2     Bond   0.1        01-Jan-2000 01-Jan-2004 2        10% bond

Index Type Strike Settle       Maturity    CapReset... Name ...
3     Cap  0.15   01-Jan-2000 01-Jan-2004 1            15% Cap
```

```
[Price, PriceTree] = bdtprice(BDTTree, BDTSubSet);

Warning: Not all cash flows are aligned with the tree. Result will
be approximated.

Price =

    95.5030
    93.9079
     1.4863
```

You can use `treeviewer` to see the prices of these three instruments along the price tree.

10% Bond

10% Bond

15% Cap

See Also   bdtsens, bdttree, instadd, intenvprice, intenvsens

bdtsens

Purpose   Fixed income instrument prices and sensitivities by BDT interest rate tree

Syntax   [Delta, Gamma, Vega, Price] = bdtsens(BDTTree, InstSet, Options)

Arguments
BDTTree	Interest rate tree structure created by bdttree.
InstSet	Variable containing a collection of NINST instruments. Instruments are categorized by type; each type can have different data fields. The stored data field is a row vector or string for each instrument.
Options	(Optional) Derivatives pricing options structure created with derivset.

Description   [Delta, Gamma, Vega, Price] = bdtsens(BDTTree, InstSet, Options)
computes instrument sensitivities and prices for instruments using an interest
rate tree created with bdttree. NINST instruments from a financial instrument
variable, InstSet, are priced. bdtsens handles instrument types: 'Bond',
'CashFlow', 'OptBond', 'Fixed', 'Float', 'Cap', 'Floor', 'Swap'. See
instadd for information on instrument types.

Delta is an NINST-by-1 vector of deltas, representing the rate of change of
instrument prices with respect to changes in the interest rate. Delta is
computed by finite differences in calls to bdttree. See bdttree for information
on the observed yield curve.

Gamma is an NINST-by-1 vector of gammas, representing the rate of change of
instrument deltas with respect to the changes in the interest rate. Gamma is
computed by finite differences in calls to bdttree.

Vega is an NINST-by-1 vector of vegas, representing the rate of change of
instrument prices with respect to the changes in the volatility $\sigma(t, T)$. Vega is
computed by finite differences in calls to bdttree. See bdtvolspec for
information on the volatility process.

Note All sensitivities are returned as dollar sensitivities. To find the
per-dollar sensitivities, divide by the respective instrument price.

Price is an NINST-by-1 vector of prices of each instrument. The prices are computed by backward dynamic programming on the interest rate tree. If an instrument cannot be priced, NaN is returned.

Delta and Gamma are calculated based on yield shifts of 100 basis points. Vega is calculated based on a 1% shift in the volatility process.

Examples   Load the tree and instruments from a data file. Compute delta and gamma for the cap and bond instruments contained in the instrument set.

```
load deriv.mat;
BDTSubSet = instselect(BDTInstSet,'Type', {'Bond', 'Cap'});

instdisp(BDTSubSet)
```

```
Index Type CouponRate Settle        Maturity     Period  Name ...
1     Bond 0.1        01-Jan-2000   01-Jan-2003  1       10% Bond
2     Bond 0.1        01-Jan-2000   01-Jan-2004  2       10% Bond

Index Type Strike Settle     Maturity     CapReset... Name ...
3     Cap  0.15   01-Jan-2000 01-Jan-2004 1           15% Cap
```

```
[Delta, Gamma] = bdtsens(BDTTree, BDTSubSet)

Warning: Not all cash flows are aligned with the tree. Result will
be approximated.

Delta =

 -232.6681
 -281.0517
   78.3776
```

```
Gamma =

  1.0e+003 *

    0.8037
    1.1819
    0.7490
```

See Also   bdtprice, bdttree, bdtvolspec, instadd

bdttimespec

Purpose	Specify time structure for BDT interest rate tree

Syntax

```
TimeSpec = bdttimespec(ValuationDate, Maturity, Compounding)
```

Arguments

ValuationDate	Scalar date marking the pricing date and first observation in the tree. Specify as serial date number or date string
Maturity	Number of levels (depth) of the tree. A number of levels (NLEVELS)-by-1 vector of dates marking the cash flow dates of the tree. Cash flows with these maturities fall on tree nodes. Maturity should be in increasing order.
Compounding	(Optional) Scalar value representing the rate at which the input zero rates were compounded when annualized. Default = 1. 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.

Description

TimeSpec = bdttimespec(ValuationDate, Maturity, Compounding) sets the number of levels and node times for a BDT tree and determines the mapping between dates and time for rate quoting.

TimeSpec is a structure specifying the time layout for bdttree. The state observation dates are [ValuationDate; Maturity(1:end-1)]. Because a forward rate is stored at the last observation, the tree can value cash flows out to Maturity.

Examples

Specify a four period tree with annual nodes. Use annual compounding to report rates.

```
Compounding = 1;
ValuationDate = '01-01-2000';
Maturity = ['01-01-2001'; '01-01-2002'; '01-01-2003';
'01-01-2004'; '01-01-2005'];
Volatility = [.2; .19; .18; .17; .16];

TimeSpec = bdttimespec(ValuationDate, Maturity, Compounding)

TimeSpec =

           FinObj: 'BDTTimeSpec'
    ValuationDate: 730486
         Maturity: [5x1 double]
      Compounding: 1
            Basis: 0
      EndMonthRule: 1
```

See Also   bdttree, bdtvolspec

Purpose   Build BDT interest rate tree

Syntax   BDTTree = bdttree(VolSpec, RateSpec, TimeSpec)

Arguments | | |
|---|---|
| VolSpec | Volatility process specification. See bdtvolspec for information on the volatility process. |
| RateSpec | Interest rate specification for the initial rate curve. See intenvset for information on declaring an interest rate variable. |
| TimeSpec | Tree time layout specification. Defines the observation dates of the BDT tree and the Compounding rule for date to time mapping and price-yield formulas. See bdttimespec for information on the tree structure. |

Description BDTTree = bdttree(VolSpec, RateSpec, TimeSpec) creates a structure containing time and interest rate information on a recombining tree.

Examples   Using the data provided, create a BDT volatility specification (VolSpec), rate specification (RateSpec), and tree time layout specification (TimeSpec). Then use these specifications to create a BDT tree with bdttree.

```
Compounding = 1;
ValuationDate = '01-01-2000';
StartDate = ValuationDate;
EndDates = ['01-01-2001'; '01-01-2002'; '01-01-2003';
'01-01-2004'; '01-01-2005'];
Rates = [.1; .11; .12; .125; .13];
Volatility = [.2; .19; .18; .17; .16];

RateSpec = intenvset('Compounding', Compounding,...
                     'ValuationDate', ValuationDate,...
                     'StartDates', StartDate,...
                     'EndDates', EndDates,...
                     'Rates', Rates);


BDTTimeSpec = bdttimespec(ValuationDate, EndDates, Compounding);
BDTVolSpec = bdtvolspec(ValuationDate, EndDates, Volatility);
```

```
BDTTree = bdttree(BDTVolSpec, RateSpec, BDTTimeSpec);
```

Use treeviewer to observe the tree you have created.

```
treeviewer(BDTTree)
```
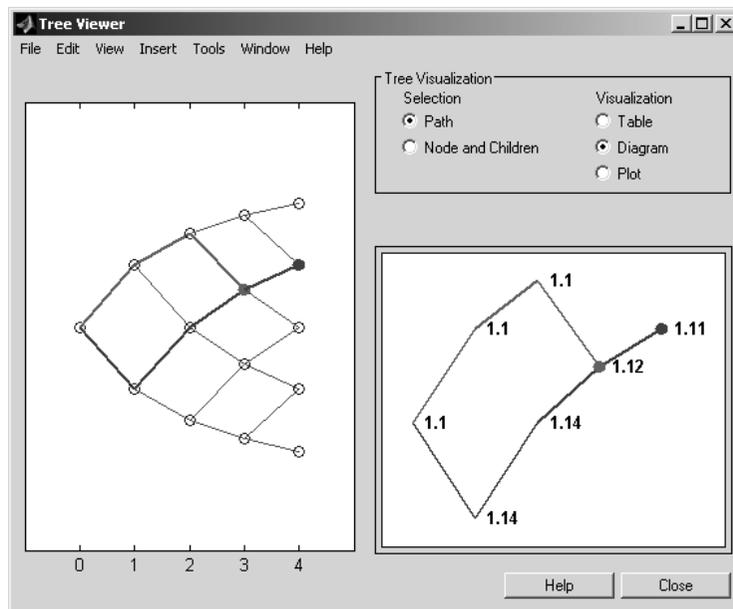

See Also   bdtprice, bdttimespec, bdtvolspec, intenvset

Purpose   Specify a BDT interest rate volatility process

Syntax   Volspec = bdtvolspec(ValuationDate, VolDates, VolCurve,
                   InterpMethod)

Arguments

ValuationDate	Scalar value representing the observation date of the investment horizon.
VolDates	Number of points (NPOINTS)-by-1 vector of yield volatility end dates.
VolCurve	NPOINTS-by-1 vector of yield volatility values in decimal form.
InterpMethod	(Optional) Interpolation method. Default is 'linear'. See interp1 for more information.

Description   Volspec = bdtvolspec(ValuationDate, VolDates, VolCurve,
                   InterpMethod) creates a structure specifying the volatility for bdttree.

Examples   Using the data provided, create a BDT volatility specification (VolSpec).

```
ValuationDate = '01-01-2000';
EndDates = ['01-01-2001'; '01-01-2002'; '01-01-2003';
'01-01-2004'; '01-01-2005'];
Volatility = [.2; .19; .18; .17; .16];

BDTVolSpec = bdtvolspec(ValuationDate, EndDates, Volatility)

BDTVolSpec =
                FinObj: 'BDTVolSpec'
         ValuationDate: 730486
              VolDates: [5x1 double]
              VolCurve: [5x1 double]
        VolInterpMethod: 'linear'
```

See Also   bdttree, interp1

bondbybdt

Purpose   Price bond by BDT interest rate tree

Syntax   [Price, PriceTree] = bondbybdt(BDTTree, CouponRate, Settle,
                    Maturity, Period, Basis, EndMonthRule, IssueDate,
                    FirstCouponDate, LastCouponDate, StartDate, Face, Options)

Arguments

BDTTree	Interest rate tree structure created by bdttree.
CouponRate	Decimal annual rate.
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 1, 2, 3, 4, 6, and 12. Default = 2.
Basis	(Optional) Day-count basis of the bond. A vector of integers. 0 = actual/actual (default), 1 = 30/360, 2 = actual/360, 3 = actual/365.
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 is followed only by the bond's maturity cash flow date.
StartDate	Ignored.
Face	(Optional) Face value. Default is `100`.
Options	(Optional) Derivatives pricing options structure created with `derivset`.

The `Settle` date for every bond is set to the `ValuationDate` of the BDT tree. The bond argument `Settle` is ignored.

Description   [Price, PriceTree] = bondbybdt(BDTTree, CouponRate, Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate, StartDate, Face, Options) computes the price of a bond by a BDT interest rate tree.

`Price` is a number of instruments (NINST)-by-1 matrix of expected prices at time 0.

`PriceTree` is a MATLAB structure of trees containing vectors of instrument prices and accrued interest, and a vector of observation times for each node. Within `PriceTree`:

- `PriceTree.PTree` contains the clean prices.
- `PriceTree.AITree` contains the accrued interest.
- `PriceTree.tObs` contains the observation times.

Examples   Price a 10% bond using a BDT interest rate tree.

Load the file `deriv.mat`, which provides `BDTTree`. `BDTTree` contains the time and interest rate information needed to price the bond.

```
load deriv
```

Set the required values. Other arguments will use defaults.

```
CouponRate = 0.10;
Settle = '01-Jan-2000';
Maturity = '01-Jan-2003';
Period = 1;
```

Use bondbybdt to compute the price of the bond.

```
Price = bondbybdt(BDTTree, CouponRate, Settle, Maturity, Period)

Price =

   95.5030
```

See Also   bdttree, bdtprice, instbond

Purpose   Price bond by HJM interest rate tree

Syntax   [Price, PriceTree] = bondbyhjm(HJMTree, CouponRate, Settle,
        Maturity, Period, Basis, EndMonthRule, IssueDate,
        FirstCouponDate, LastCouponDate, StartDate, Face, Options)

Arguments

HJMTree	Forward rate tree structure created by hjmtree.
CouponRate	Decimal annual rate.
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 1, 2, 3, 4, 6, and 12. Default = 2.
Basis	(Optional) Day-count basis of the bond. A vector of integers. 0 = actual/actual (default), 1 = 30/360, 2 = actual/360, 3 = actual/365.
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 is followed only by the bond's maturity cash flow date.
StartDate	Ignored.
Face	(Optional) Face value. Default = 100.
Options	(Optional) Derivatives pricing options structure created with derivset.

The Settle date for every bond is set to the ValuationDate of the HJM tree. The bond argument Settle is ignored.

Description [Price, PriceTree] = bondbyhjm(HJMTree, CouponRate, Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate, StartDate, Face, Options) computes the price of a bond by an HJM forward rate tree.

Price is a number of instruments (NINST)-by-1 matrix of expected prices at time 0.

PriceTree is a structure of trees containing vectors of instrument prices and accrued interest, and a vector of observation times for each node. Within PriceTree:

- PriceTree.PBush contains the clean prices.
- PriceTree.AIBush contains the accrued interest.
- PriceTree.tObs contains the observation times.

Examples Price a 4% bond using an HJM forward rate tree.

Load the file deriv.mat, which provides HJMTree. HJMTree contains the time and forward rate information needed to price the bond.

```
load deriv
```

Set the required values. Other arguments will use defaults.

```
CouponRate = 0.04;
Settle = '01-Jan-2000';
Maturity = '01-Jan-2004';
```

Use bondbyhjm to compute the price of the bond.

```
Price = bondbyhjm(HJMTree, CouponRate, Settle, Maturity)
Warning: Not all cash flows are aligned with the tree. Result will
be approximated.

Price =

  97.5280
```

See Also   hjmtree, hjmprice, instbond

bondbyzero

Purpose   Price bond by a set of zero curves

Syntax   Price = bondbyzero(RateSpec, CouponRate, Settle, Maturity, Period,
                 Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate,
                 StartDate, Face)

Arguments

RateSpec	A structure encapsulating the properties of an interest rate structure. See intenvset for information on creating RateSpec.
CouponRate	Decimal annual rate.
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 1, 2, 3, 4, 6, and 12. Default = 2.
Basis	(Optional) Day-count basis of the bond. A vector of integers. 0 = actual/actual (default), 1 = 30/360, 2 = actual/360, 3 = actual/365.
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 is followed only by the bond's maturity cash flow date.
StartDate	Ignored.
Face	(Optional) Face value. Default = 100.

All inputs are either scalars or number of instruments (NINST)-by-1 vectors unless otherwise specified. Dates can be serial date numbers or date strings. Optional arguments can be passed as empty matrix [].

Description   Price = bondbyzero(RateSpec, CouponRate, Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate, StartDate, Face) returns a NINST-by-NUMCURVES matrix of clean bond prices. Each column arises from one of the zero curves.

Examples   Price a 4% bond using a set of zero curves.

Load the file deriv.mat, which provides ZeroRateSpec, the interest rate term structure needed to price the bond.

```
load deriv
```

Set the required values. Other arguments will use defaults.

```
CouponRate = 0.04;
Settle = '01-Jan-2000';
Maturity = '01-Jan-2004';
```

Use bondbyzero to compute the price of the bond.

```
Price = bondbyzero(ZeroRateSpec, CouponRate, Settle, Maturity)

Price =

  97.5334
```

See Also   cfbyzero, fixedbyzero, floatbyzero, swapbyzero

Purpose	Extract entries from node of bushy tree

Syntax   Values = bushpath(Tree, BranchList)

Arguments

Tree	Bushy tree.
BranchList	Number of paths (NUMPATHS) by path length (PATHLENGTH) matrix containing the sequence of branchings.

Description   Values = bushpath(Tree, BranchList) extracts entries of a node of a bushy tree. The node path is described by the sequence of branchings taken, starting at the root. The top branch is number one, the second-to-top is two, and so on. Set the branch sequence to zero to obtain the entries at the root node.

Values is a number of values (NUMVALS)-by-NUMPATHS matrix containing the retrieved entries of a bushy tree.

Examples   Create an HJM tree by loading the example file.

```
load deriv.mat;
```

Then

```
FwdRates = bushpath(HJMTree.FwdTree, [1 2 1])
```

returns the rates at the tree nodes located by taking the up branch, then the down branch, and finally the up branch again.

```
FwdRates =

    1.0356
    1.0364
    1.0526
    1.0463
```

You can visualize this with the treeviewer function.

bushpath

```
treeviewer(HJMTree)
```
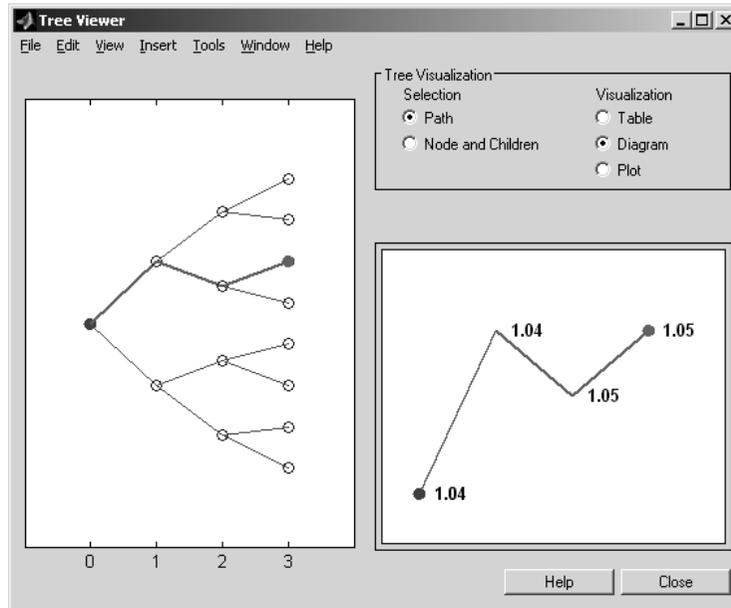

See Also   bushshape, mkbush

Purpose   Retrieve shape of bushy tree

Syntax   [NumLevels, NumChild, NumPos, NumStates, Trim] = bushshape(Tree)

Arguments   Tree   Bushy tree.

Description   [NumLevels, NumChild, NumPos, NumStates, Trim] = bushshape(Tree)
                   returns information on a bushy tree's shape.

                   NumLevels is the number of time levels of the tree.

                   NumChild is a 1 by number of levels (NUMLEVELS) vector with the number of
                   branches (children) of the nodes in each level.

                   NumPos is a 1-by-NUMLEVELS vector containing the length of the state vectors in
                   each level.

                   NumStates is a 1-by-NUMLEVELS vector containing the number of state vectors in
                   each level.

                   Trim is 1 if NumPos decreases by one when moving from one time level to the
                   next. Otherwise, it is 0.

Examples   Create an HJM tree by loading the example file.

                       load deriv.mat;

                   With treeviewer you can see the general shape of the HJM interest rate tree.
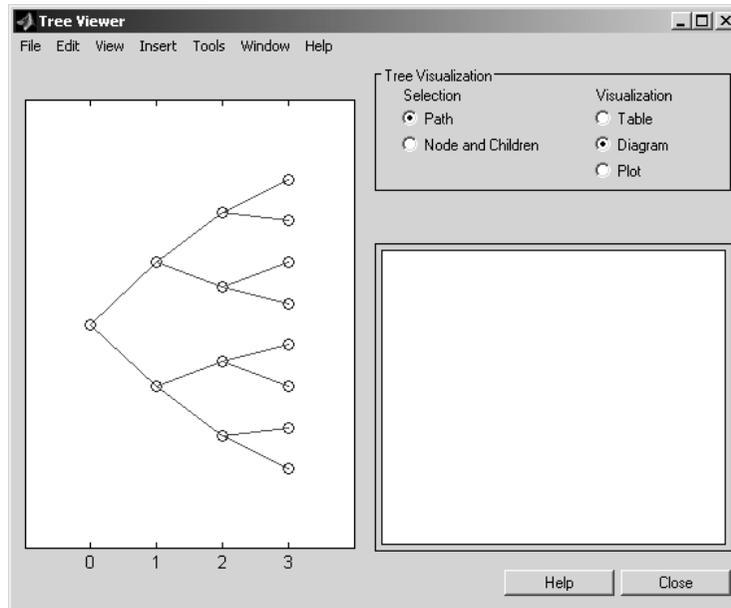
bushshape

With this tree

```
[NumLevels, NumChild, NumPos, NumStates, Trim] =...
bushshape(HJMTree.FwdTree)
```

returns

```
NumLevels =
     4

NumChild =
     2    2    2    0

NumPos =
     4    3    2    1

NumStates =
     1    2    4    8

Trim =
     1
```

You can recreate this tree using the mkbush function.

```
Tree = mkbush(NumLevels, NumChild(1), NumPos(1), Trim);
Tree = mkbush(NumLevels, NumChild, NumPos);
```

See Also   bushpath, mkbush

capbybdt

Purpose	Price cap instrument by BDT interest rate tree
Syntax	[Price, PriceTree] = capbybdt(BDTTree, Strike, Settle, Maturity, Reset, Basis, Principal, Options)

Arguments

BDTTree	Interest rate tree structure created by bdttree.
Strike	Number of instruments (NINST)-by-1 vector of rates at which the cap is exercised.
Settle	Settlement dates. NINST-by-1 vector of dates representing the settlement dates of the cap.
Maturity	NINST-by-1 vector of dates representing the maturity dates of the cap.
Reset	(Optional) NINST-by-1 vector representing the frequency of payments per year. Default = 1.
Basis	(Optional) NINST-by-1 vector representing the basis used when annualizing the input forward rate tree. Default = 0 (actual/actual).
Principal	(Optional) The notional principal amount. Default = 100.
Options	(Optional) Derivatives pricing options structure created with derivset.

Description   [Price, PriceTree] = capbybdt(BDTTree, Strike, Settle, Maturity, Reset, Basis, Principal, Options) computes the price of a cap instrument from a BDT interest rate tree.

Price is the expected price of the cap at time 0.

PriceTree is the tree structure with values of the cap at each node.

The Settle date for every cap is set to the ValuationDate of the BDT tree. The cap argument Settle is ignored.

Examples   Example 1.

Price a 3% cap instrument using a BDT interest rate tree.

Load the file `deriv.mat`, which provides `BDTTree`. `BDTTree` contains the time and interest rate information needed to price the cap instrument.

```
load deriv
```

Set the required values. Other arguments will use defaults.

```
Strike = 0.03;
Settle = '01-Jan-2000';
Maturity = '01-Jan-2004';
```

Use `capbybdt` to compute the price of the cap instrument.

```
Price = capbybdt(BDTTree, Strike, Settle, Maturity)

Price =

   28.5191
```

Example 2.

Here is a second example, showing the pricing of a 10% cap instrument using a newly-created BDT tree.

First set the required arguments for the three needed specifications.

```
Compounding = 1;
ValuationDate = '01-01-2000';
StartDate = ValuationDate;
EndDates = ['01-01-2001'; '01-01-2002'; '01-01-2003';
'01-01-2004'; '01-01-2005'];
Rates = [.1; .11; .12; .125; .13];
Volatility = [.2; .19; .18; .17; .16];
```

Next create the specifications.

```
RateSpec = intenvset('Compounding', Compounding,...
'ValuationDate', ValuationDate,...
'StartDates', StartDate,...
'EndDates', EndDates,...
'Rates', Rates);
BDTTimeSpec = bdttimespec(ValuationDate, EndDates, Compounding);
BDTVolSpec = bdtvolspec(ValuationDate, EndDates, Volatility);
```

Now create the BDT tree from the specifications.

```
BDTTree = bdttree(BDTVolSpec, RateSpec, BDTTimeSpec);
```

Set the cap arguments.

```
CapStrike = 0.10;
Settlement = ValuationDate;
Maturity = '01-01-2002';
CapReset = 1;
```

Remaining arguments will use defaults.

Finally, use capbybdt to find the price of the cap instrument.

```
Price= capbybdt(BDTTree, CapStrike, Settlement, Maturity,...
CapReset)

Price =

    1.6923
```

See Also   bdttree, cfbybdt, floorbybdt, swapbybdt

Purpose	Price cap instrument by HJM interest rate tree

Syntax

```
[Price, PriceTree] = capbyhjm(HJMTree, Strike, Settle, Maturity,
    Reset, Basis, Principal, Options)
```

Arguments

HJMTree	Forward rate tree structure created by hjmtree.
Strike	Number of instruments (NINST)-by-1 vector of rates at which the cap is exercised.
Settle	Settlement dates. NINST-by-1 vector of dates representing the settlement dates of the cap.
Maturity	NINST-by-1 vector of dates representing the maturity dates of the cap.
Reset	(Optional) NINST-by-1 vector representing the frequency of payments per year. Default = 1.
Basis	(Optional) NINST-by-1 vector representing the basis used when annualizing the input forward rate tree. Default = 0 (actual/actual).
Principal	(Optional) The notional principal amount. Default = 100.
Options	(Optional) Derivatives pricing options structure created with derivset.

Description

[Price, PriceTree] = capbyhjm(HJMTree, Strike, Settle, Maturity, Reset, Basis, Principal, Options) computes the price of a cap instrument from an HJM tree.

Price is the expected price of the cap at time 0.

PriceTree is the tree structure with values of the cap at each node.

The Settle date for every cap is set to the ValuationDate of the HJM tree. The cap argument Settle is ignored.

Examples

Price a 3% cap instrument using an HJM forward rate tree.

Load the file deriv.mat, which provides HJMTree. HJMTree contains the time and forward rate information needed to price the cap instrument.

```
load deriv
```

Set the required values. Other arguments will use defaults.

```
Strike = 0.03;
Settle = '01-Jan-2000';
Maturity = '01-Jan-2004';
```

Use capbyhjm to compute the price of the cap instrument.

```
Price = capbyhjm(HJMTree, Strike, Settle, Maturity)

Price =

  6.2831
```

See Also   cfbyhjm, floorbyhjm, hjmtree, swapbyhjm

Purpose	Price cash flows from BDT interest rate tree
Syntax	[Price, PriceTree] = cfbybdt(BDTTree, CFlowAmounts, CFlowDates, Settle, Basis, Options)

Arguments

BDTTree	Forward rate tree structure created by bdttree.
CFlowAmounts	Number of instruments (NINST) by maximum number of cash flows (MOSTCFS) matrix of cash flow amounts. Each row is a list of cash flow values for one instrument. If an instrument has fewer than MOSTCFS cash flows, the end of the row is padded with NaNs.
CFlowDates	NINST-by-MOSTCFS matrix of cash flow dates. Each entry contains the date of the corresponding cash flow in CFlowAmounts.
Settle	Settlement date. A vector of serial date numbers or date strings. The Settle date for every cash flow is set to the ValuationDate of the HJM tree. The cash flow argument, Settle, is ignored.
Basis	(Optional) Day-count basis of the bond. A vector of integers. 0 = actual/actual (default), 1 = 30/360, 2 = actual/360, 3 = actual/365.
Options	(Optional) Derivatives pricing options structure created with derivset.

Description   [Price, PriceTree] = cfbybdt(BDTTree, CFlowAmounts, CFlowDates, Settle, Basis, Options) prices cash flows from a BDT interest rate tree.

Price is an NINST-by-1 vector of expected prices at time 0.

PriceTree is a tree structure with a vector of instrument prices at each node.

Examples   Price a portfolio containing two cash flow instruments paying interest annually over the four year period from January 1, 2000 to January 1, 2004.

Load the file deriv.mat, which provides BDTTree. BDTTree contains the time and interest rate information needed to price the instruments.

```
load deriv
CFlowAmounts =[5 NaN 5.5 105;5 0 6 105];
CFlowDates = [730852, NaN, 731582,731947;
              730852, 731217, 731582, 731947];

Price = cfbybdt(BDTTree, CFlowAmounts, CFlowDates,...
BDTTree.RateSpec.ValuationDate)

Price =

   74.0112
   74.3671

PriceTree =

    FinObj: 'BDTPriceTree'
      tObs: [0 1.00 2.00 3.00 4.00]
     PTree: {1x5 cell}
```

You can visualize the prices of the two cash flow instruments with the treeviewer function.

```
treeviewer(PriceTree)
```
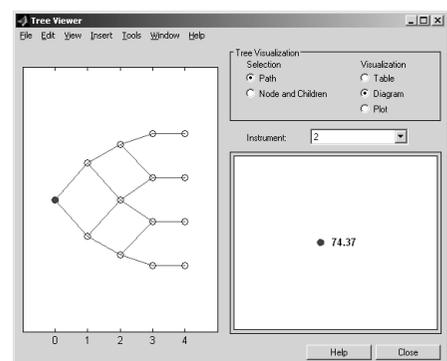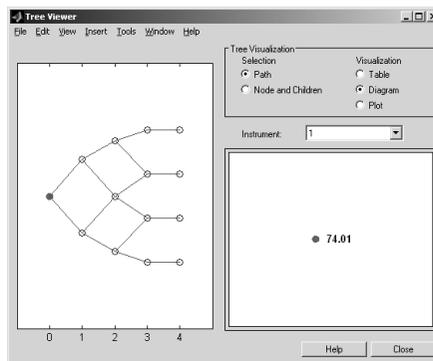

See Also   bdttree, bdtprice, cfamounts, instcf

Purpose	Price cash flows from HJM interest rate tree
Syntax	`[Price, PriceTree] = cfbyhjm(HJMTree, CFlowAmounts, CFlowDates, Settle, Basis, Options)`

Arguments

HJMTree	Forward rate tree structure created by hjmtree.
CFlowAmounts	Number of instruments (NINST) by maximum number of cash flows (MOSTCFS) matrix of cash flow amounts. Each row is a list of cash flow values for one instrument. If an instrument has fewer than MOSTCFS cash flows, the end of the row is padded with NaNs.
CFlowDates	NINST-by-MOSTCFS matrix of cash flow dates. Each entry contains the date of the corresponding cash flow in CFlowAmounts.
Settle	Settlement date. A vector of serial date numbers or date strings. The Settle date for every cash flow is set to the ValuationDate of the HJM tree. The cash flow argument, Settle, is ignored.
Basis	(Optional) Day-count basis of the bond. A vector of integers. 0 = actual/actual (default), 1 = 30/360, 2 = actual/360, 3 = actual/365.
Options	(Optional) Derivatives pricing options structure created with derivset.

Description   `[Price, PriceTree] = cfbyhjm(HJMTree, CFlowAmounts, CFlowDates, Settle, Basis, Options)` prices cash flows from an HJM interest rate tree.

Price is an NINST-by-1 vector of expected prices at time 0.

PriceTree is a tree structure with a vector of instrument prices at each node.

Examples   Price a portfolio containing two cash flow instruments paying interest annually over the four year period from January 1, 2000 to January 1, 2004.

Load the file deriv.mat, which provides HJMTree. HJMTree contains the time and forward rate information needed to price the instruments.

```
load deriv
CFlowAmounts =[5 NaN 5.5 105;5 0 6 105];
CFlowDates = [730852, NaN, 731582,731947;
              730852, 731217, 731582, 731947];

[Price, PriceTree] = cfbyhjm(HJMTree, CFlowAmounts,...
CFlowDates, HJMTree.RateSpec.ValuationDate)

Price =

    96.7805
    97.2188
PriceTree =

    FinObj: 'HJMPriceTree'
      tObs: [0 1.00 2.00 3.00 4.00]
     PBush: {1x5 cell}
```

You can visualize the prices of the two cash flow instruments with the treeviewer function.

```
treeviewer(PriceTree)
```
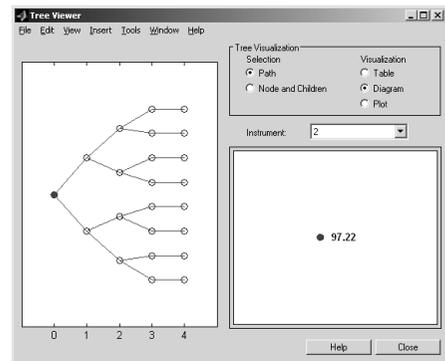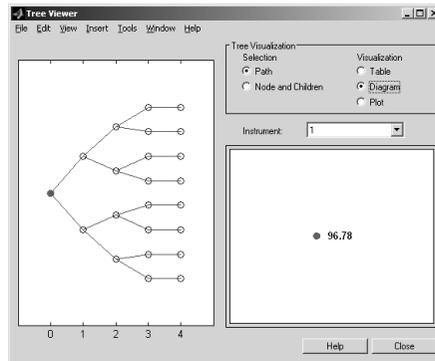

See Also   cfamounts, hjmprice, hjmtree, instcf

Purpose	Price cash flows by a set of zero curves

Syntax   Price = cfbyzero(RateSpec, CFlowAmounts, CFlowDates, Settle, Basis)

Arguments

RateSpec	A structure encapsulating the properties of an interest rate structure. See intenvset for information on creating RateSpec.
CFlowAmounts	Number of instruments (NINST) by maximum number of cash flows (MOSTCFS) matrix with entries listing cash flow amounts corresponding to each date in CFlowDates. Each row is a list of cash flow values for one instrument. If an instrument has fewer than MOSTCFS cash flows, the end of the row is padded with NaNs.
CFlowDates	NINST-by-MOSTCFS matrix of cash flow dates. Each entry contains the serial date of the corresponding cash flow in CFlowAmounts.
Settle	Settlement date on which the cash flows are priced.
Basis	(Optional) Day-count basis of the bond. A vector of integers. 0 = actual/actual (default), 1 = 30/360, 2 = actual/360, 3 = actual/365.

Description   Price = cfbyzero(RateSpec, CFlowAmounts, CFlowDates, Settle, Basis) computes Price, an NINST-by-NUMCURVES matrix of cash flows prices. Each column arises from one of the zero curves.

Examples   Price a portfolio containing two cash flow instruments paying interest annually over the four year period from January 1, 2000 to January 1, 2004.

Load the file deriv.mat, which provides ZeroRateSpec. ZeroRateSpec contains the interest rate information needed to price the instruments.

```
load deriv
CFlowAmounts =[5 NaN 5.5 105;5 0 6 105];
CFlowDates = [730852, NaN, 731582,731947;
              730852, 731217, 731582, 731947];
Settle = 730486;
```

```
Price = cfbyzero(ZeroRateSpec, CFlowAmounts, CFlowDates, Settle)

Price =

   96.7804
   97.2187
```

See Also   bondbyzero, fixedbyzero, floatbyzero, swapbyzero

Purpose	Create financial structure or return financial structure class name

Syntax	`Obj = classfin(ClassName)`
	`Obj = classfin(Struct, ClassName)`
	`ClassName = classfin(Obj)`

Arguments

ClassName	String containing name of financial structure class.
Struct	MATLAB structure to be converted into a financial structure.
Obj	Name of a financial structure.

Description   `Obj = classfin(ClassName)` and `Obj = classfin(Struct, ClassName)` create a financial structure of class `ClassName`.

`ClassName = classfin(Obj)` returns a string containing a financial structure's class name.

Examples   Example 1.

Create an `HJMTimeSpec` financial structure and complete its fields. (Typically, the function `hjmtimespec` is used to create `HJMTimeSpec` structures).

```
TimeSpec = classfin('HJMTimeSpec');
TimeSpec.ValuationDate = datenum('Dec-10-1999');
TimeSpec.Maturity = datenum('Dec-10-2002');
TimeSpec.Compounding = 2;
TimeSpec.Basis = 0;
TimeSpec.EndMonthRule = 1;

TimeSpec =

            FinObj: 'HJMTimeSpec'
     ValuationDate: 730464
          Maturity: 731560
       Compounding: 2
             Basis: 0
       EndMonthRule: 1
```

Example 2.

Convert an existing MATLAB structure into a financial structure.

```
TSpec.ValuationDate = datenum('Dec-10-1999');
TSpec.Maturity = datenum('Dec-10-2002');
TSpec.Compounding = 2;
TSpec.Basis = 0;
TSpec.EndMonthRule = 0;

TimeSpec = classfin(TSpec, 'HJMTimeSpec')

TimeSpec =

    ValuationDate: 730464
         Maturity: 731560
      Compounding: 2
            Basis: 0
      EndMonthRule: 0
            FinObj: 'HJMTimeSpec'
```

Example 3.

Obtain a financial structure's class name.

```
load deriv.mat
ClassName = classfin(HJMTree)

ClassName =

HJMFwdTree
```

See Also   isafin

Purpose	Fixed income time and frequency from dates	

Syntax

```
[Times, F] = date2time(Settle, Maturity, Compounding, Basis,
    EndMonthRule)
```

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.
Compounding	Scalar value representing the rate at which the input zero rates were compounded when annualized. This argument determines the formula for the discount factors: Compounding = 1, 2, 3, 4, 6, 12 Disc = $(1 + Z/F)\wedge(-T)$, 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 = $(1 + Z/F)\wedge(-T)$, 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 bond. A vector of integers. 0 = actual/actual (default), 1 = 30/360, 2 = actual/360, 3 = actual/365.
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.

date2time

Description   `[Times, F] = date2time(Settle, Dates, Compounding, Basis, EndMonthRule)` computes time factors appropriate to compounded rate quotes between `Settle` and `Maturity` dates.

`Times` is a vector of time factors.

`F` is a scalar of related compounding frequencies.

See Also   `cftimes` in the *Financial Toolbox User's Guide*

`disc2rate`, `rate2disc`

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 the 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   datenum, datestr in the *Financial Toolbox User's Guide*

Remarks   This function is identical to the datedisp function in the Financial Toolbox.

derivget

Purpose	Get derivatives pricing options
Syntax	Value = derivget(Options, '*Parameter*')

Arguments

Options	Existing options specification structure, probably created from previous call to derivset.
Parameter	Must be 'Diagnostics', 'Warnings', or 'ConstRate'. It is sufficient to type only the leading characters that uniquely identify the parameter. Case is ignored for parameter names.

Description   Value = derivget(Options, '*Parameter*') extracts the value of the named parameter from the derivative options structure Options. Parameter values can be 'off' or 'on'.

Examples   Create an Options structure with the value of Diagnostics set to 'on'.

```
Options = derivset('Diagnostics','on')
```

Use derivget to extract the value of Diagnostics from the Options structure.

```
Value = derivget(Options, 'Diagnostics')

Value =

on
```

Use derivget to extract the value of ConstRate.

```
Value   = derivget(Options, 'ConstRate')

Value =

on
```

Because the value of 'ConstRate' was not previously set with derivset, the answer represents the default setting for 'ConstRate'.

See Also   derivset

Purpose	Set or modify derivatives pricing options

Syntax

```
Options = derivset(Options, 'Parameter1', Value1, 'Parameter2',
    Value2,'Parameter3', Value3)
Options = derivset(OldOptions, NewOptions)
Options = derivset
derivset
```

Arguments

Options	(Optional) Existing options specification structure, probably created from previous call to derivset.
Parameter*n* Value*n*	Must be 'Diagnostics', 'Warnings', or 'ConstRate'. Parameters can be entered in any order. Parameter values can be 'on' or 'off'. 'Diagnostics' 'on' generates diagnostic information. The default is 'Diagnostics' 'off'. 'Warnings' 'on' (default) displays a warning message when executing a pricing function. 'ConstRate' 'on' (default) assumes a constant rate between tree nodes.
OldOptions	Existing options specification structure.
NewOptions	New options specification structure.

Description Options = derivset(Options, 'Parameter1', Value1, 'Parameter2', Value2,'Parameter3', Value3) creates a derivatives pricing options structure Options in which the named parameters have the specified values. Any unspecified value is set to the default value for that parameter when Options is passed to the pricing function. It is sufficient to type only the leading characters that uniquely identify the parameter name. Case is also ignored for parameter names.

If the optional input argument Options is specified, derivset modifies an existing pricing options structure by changing the named parameters to the specified values.

derivset

Note For parameter *values*, correct case and the complete string are required; if an invalid string is provided, the default is used.

Options = derivset(OldOptions, NewOptions) combines an existing options structure OldOptions with a new options structure NewOptions. Any parameters in NewOptions with nonempty values overwrite the corresponding old parameters in OldOptions.

Options = derivset creates an options structure Options whose fields are set to the default values.

derivset with no input or output arguments displays all parameter names and information about their possible values.

Examples
```
Options = derivset('Diagnostics','on')
```

enables the display of additional diagnostic information that appears when executing pricing functions.

```
Options = derivset(Options, 'ConstRate', 'off')
```

changes the ConstRate parameter in the existing Options structure so that the assumption of constant rates between tree nodes no longer applies.

With no input or output arguments derivset displays all parameter names and information about their possible values.

```
derivset
            Diagnostics: [ on   | {off} ]
               Warnings: [ {on} | off   ]
              ConstRate: [ {on} | off   ]
```

See Also   derivget

4-54

Purpose   Interest rates from cash flow discounting factors

Syntax   Usage 1: Interval points are input as times in periodic units.

                 Rates = disc2rate(Compounding, Disc, EndTimes, StartTimes)

                 Usage 2: ValuationDate is passed and interval points are input as dates.

                 [Rates, EndTimes, StartTimes] = disc2rate(Compounding, Disc,
                   EndDates, StartDates, ValuationDate)

Arguments   Compounding   Scalar value representing the rate at which the input
                                      zero rates were compounded when annualized. This
                                      argument determines the formula for the discount
                                      factors:
                                      Compounding = 1, 2, 3, 4, 6, 12
                                      Disc = $(1 + Z/F)^{(-T)}$, where F is the compounding
                                              frequency, Z is the zero rate, and T is the time in
                                              periodic units, e.g. T = F is one year.
                                      Compounding = 365
                                      Disc = $(1 + Z/F)^{(-T)}$, where F is the number of days
                                              in the basis year and T is a number of days
                                              elapsed computed by basis.
                                      Compounding = -1
                                      Disc = $\exp(-T*Z)$, where T is time in years.

                 Disc   Number of points (NPOINTS) by number of curves
                                      (NCURVES) matrix of discounts. Disc are unit bond prices
                                      over investment intervals from StartTimes, when the
                                      cash flow is valued, to EndTimes, when the cash flow is
                                      received.

                 EndTimes   NPOINTS-by-1 vector or scalar of times in periodic units
                                      ending the interval to discount over.

                 StartTimes   (Optional) NPOINTS-by-1 vector or scalar of times in
                                      periodic units starting the interval to discount over.
                                      Default = 0.

EndDates	NPOINTS-by-1 vector or scalar of serial maturity dates ending the interval to discount over.
StartDates	(Optional) NPOINTS-by-1 vector or scalar of serial dates starting the interval to discount over. Default = ValuationDate.
ValuationDate	Scalar value in serial date number form representing the observation date of the investment horizons entered in StartDates and EndDates. Required in Usage 2. Omitted or passed as an empty matrix to invoke Usage 1.

Description   Rates = disc2rate(Compounding, Disc, EndTimes, StartTimes) and [Rates, EndTimes, StartTimes] = disc2rate(Compounding, Disc, EndDates, StartDates, ValuationDate) convert cash flow discounting factors to interest rates. disc2rate computes the yields over a series of NPOINTS time intervals given the cash flow discounts over those intervals. NCURVES different rate curves can be translated at once if they have the same time structure. The time intervals can represent a zero curve or a forward curve.

Rates is an NPOINTS-by-NCURVES column vector of yields in decimal form over the NPOINTS time intervals.

StartTimes is an NPOINTS-by-1 column vector of times starting the interval to discount over, measured in periodic units.

EndTimes is an NPOINTS-by-1 column vector of times ending the interval to discount over, measured in periodic units.

If Compounding = 365 (daily), StartTimes and EndTimes are measured in days. The arguments otherwise contain values, T, computed from SIA semiannual time factors, Tsemi, by the formula $T = Tsemi/2 * F$, where F is the compounding frequency.

The investment intervals can be specified either with input times (Usage 1) or with input dates (Usage 2). Entering ValuationDate invokes the date interpretation; omitting ValuationDate invokes the default time interpretations.

See Also   rate2disc, ratetimes

Purpose	Price fixed rate note from BDT interest rate tree

Syntax

```
[Price, PriceTree] = fixedbybdt(BDTTree, CouponRate, Settle,
    Maturity, Reset, Basis, Principal, Options)
```

Arguments

BDTTree	Interest rate tree structure created by bdttree.
CouponRate	Decimal annual rate.
Settle	Settlement dates. Number of instruments (NINST)-by-1 vector of dates representing the settlement dates of the fixed rate note.
Maturity	NINST-by-1 vector of dates representing the maturity dates of the fixed rate note.
Reset	(Optional) NINST-by-1 vector representing the frequency of payments per year. Default = 1.
Basis	(Optional) NINST-by-1 vector representing the basis used when annualizing the input forward rate tree. Default = 0 (actual/actual).
Principal	(Optional) The notional principal amount. Default = 100.
Options	(Optional) Derivatives pricing options structure created with derivset.

Description

[Price, PriceTree] = fixedbybdt(HJMTree, CouponRate, Settle, Maturity, Reset, Basis, Principal, Options) computes the price of a fixed rate note from a BDT interest rate tree.

Price is an NINST-by-1 vector of expected prices of the fixed rate note at time 0.

PriceTree is a structure of trees containing vectors of instrument prices and accrued interest, and a vector of observation times for each node.

PriceTree.PTree contains the clean prices.

PriceTree.AITree contains the accrued interest.

PriceTree.tObs contains the observation times.

The Settle date for every fixed rate note is set to the ValuationDate of the BDT tree. The fixed rate note argument Settle is ignored.

fixedbybdt

Examples

Price a 10% fixed rate note using a BDT interest rate tree.

Load the file `deriv.mat`, which provides `BDTTree`. `BDTTree` contains the time and interest rate information needed to price the note.

```
load deriv
```

Set the required values. Other arguments will use defaults.

```
CouponRate = 0.10;
Settle = '01-Jan-2000';
Maturity = '01-Jan-2004';
Reset = 1;
```

Use `fixedbybdt` to compute the price of the note.

```
Price = fixedbybdt(BDTTree, CouponRate, Settle, Maturity, Reset)

Price =

  92.9974
```

See Also

bdttree, bondbybdt, capbybdt, cfbybdt, floatbybdt, floorbybdt, swapbybdt

Purpose　　　　Price fixed rate note from HJM interest rate tree

Syntax　　　　`[Price, PriceTree] = fixedbyhjm(HJMTree, CouponRate, Settle,`
　　　　　　　　　`Maturity, Reset, Basis, Principal, Options)`

Arguments

HJMTree	Forward rate tree structure created by hjmtree.
CouponRate	Decimal annual rate.
Settle	Settlement dates. Number of instruments (NINST)-by-1 vector of dates representing the settlement dates of the fixed rate note.
Maturity	NINST-by-1 vector of dates representing the maturity dates of the fixed rate note.
Reset	(Optional) NINST-by-1 vector representing the frequency of payments per year. Default = 1.
Basis	(Optional) NINST-by-1 vector representing the basis used when annualizing the input forward rate tree. Default = 0 (actual/actual).
Principal	(Optional) The notional principal amount. Default = 100.
Options	(Optional) Derivatives pricing options structure created with derivset.

Description　　`[Price, PriceTree] = fixedbyhjm(HJMTree, CouponRate, Settle, Maturity, Reset, Basis, Principal, Options)` computes the price of a fixed rate note from a HJM forward rate tree.

Price is an NINST-by-1 vector of expected prices of the fixed rate note at time 0.

PriceTree is a structure of trees containing vectors of instrument prices and accrued interest, and a vector of observation times for each node.

PriceTree.PBush contains the clean prices.

PriceTree.AIBush contains the accrued interest.

PriceTree.tObs contains the observation times.

The Settle date for every fixed rate note is set to the ValuationDate of the HJM tree. The fixed rate note argument Settle is ignored.

fixedbyhjm

Examples   Price a 4% fixed rate note using an HJM forward rate tree.

Load the file `deriv.mat`, which provides HJMTree. HJMTree contains the time and forward rate information needed to price the note.

```
load deriv
```

Set the required values. Other arguments will use defaults.

```
CouponRate = 0.04;
Settle = '01-Jan-2000';
Maturity = '01-Jan-2003';
```

Use `fixedbyhjm` to compute the price of the note.

```
Price = fixedbyhjm(HJMTree, CouponRate, Settle, Maturity)

Price =

  98.7159
```

See Also   bondbyhjm, capbyhjm, cfbyhjm, floatbyhjm, floorbyhjm, hjmtree, swapbyhjm

Purpose   Price fixed rate note by a set of zero curves

Syntax   Price = fixedbyzero(RateSpec, CouponRate, Settle, Maturity, Reset,
                  Basis, Principal)

Arguments

RateSpec	A structure encapsulating the properties of an interest rate structure. See intenvset for information on creating RateSpec.
CouponRate	Decimal annual rate.
Settle	Settlement date. Settle must be earlier than or equal to Maturity.
Maturity	Maturity date.
Reset	(Optional) Frequency of payments per year. Default = 1.
Basis	(Optional) Day count basis. Default = 0 (actual/actual).
Principal	(Optional) The notional principal amount. Default = 100.

All inputs are either scalars or NINST-by-1 vectors unless otherwise
specified. Any date may be a serial date number or date string. An optional
argument may be passed as an empty matrix [].

Description   Price = fixedbyzero(RateSpec, CouponRate, Settle, Maturity, Reset,
Basis, Principal) computes the price of a fixed rate note by a set of zero
curves.

Price is a number of instruments (NINST) by number of curves (NUMCURVES)
matrix of fixed rate note prices. Each column arises from one of the zero curves.

Examples   Price a 4% fixed rate note using a set of zero curves.

Load the file deriv.mat, which provides ZeroRateSpec, the interest rate term
structure needed to price the note.

    load deriv

Set the required values. Other arguments will use defaults.

```
CouponRate = 0.04;
Settle = '01-Jan-2000';
Maturity = '01-Jan-2003';
```

Use fixedbyzero to compute the price of the note.

```
Price = fixedbyzero(ZeroRateSpec, CouponRate, Settle, Maturity)

Price =

   98.7159
```

See Also   bondbyzero, cfbyzero, floatbyzero, swapbyzero

Purpose   Price floating rate note from BDT interest rate tree

Syntax   [Price, PriceTree] = floatbybdt(BDTTree, Spread, Settle, Maturity,
                         Reset, Basis, Principal, Options)

Arguments

BDTTree	Interest rate tree structure created by bdttree.
Spread	Number of instruments (NINST)-by-1 vector of number of basis points over the reference rate.
Settle	Settlement dates. NINST-by-1 vector of dates representing the settlement dates of the floating rate note.
Maturity	NINST-by-1 vector of dates representing the maturity dates of the floating rate note.
Reset	(Optional) NINST-by-1 vector representing the frequency of payments per year. Default = 1.
Basis	(Optional) NINST-by-1 vector representing the basis used when annualizing the input forward rate tree. Default = 0 (actual/actual).
Principal	(Optional) NINST-by-1 vector of the notional principal amount. Default = 100.
Options	(Optional) Derivatives pricing options structure created with derivset.

Description   [Price, PriceTree] = floatbyhjm(HJMTree, Spread, Settle, Maturity,
                       Reset, Basis, Principal, Options) computes the price of a floating rate
                       note from a BDT tree.

                       Price is an NINST-by-1 vector of expected prices of the floating rate note at time
                       0.

                       PriceTree is a structure of trees containing vectors of instrument prices and
                       accrued interest, and a vector of observation times for each node.

                       PriceTree.PTree contains the clean prices.

                       PriceTree.AITree contains the accrued interest.

PriceTree.tObs contains the observation times.

The Settle date for every floating rate note is set to the ValuationDate of the BDT tree. The floating rate note argument Settle is ignored.

Examples   Price a 20 basis point floating rate note using a BDT interest rate tree.

Load the file deriv.mat, which provides BDTTree. BDTTree contains the time and interest rate information needed to price the note.

```
load deriv
```

Set the required values. Other arguments will use defaults.

```
Spread = 20;
Settle = '01-Jan-2000';
Maturity = '01-Jan-2003';
```

Use floatbybdt to compute the price of the note.

```
Price = floatbybdt(BDTTree, Spread, Settle, Maturity)

Price =

   100.6054
```

See Also   bdttree, bondbybdt, capbybdt, cfbybdt, fixedbybdt, floorbybdt, swapbybdt

floatbyhjm

Purpose	Price floating rate note from HJM interest rate tree

Syntax

```
[Price, PriceTree] = floatbyhjm(HJMTree, Spread, Settle, Maturity,
    Reset, Basis, Principal, Options)
```

Arguments

HJMTree	Forward rate tree structure created by hjmtree.
Spread	Number of instruments (NINST)-by-1 vector of number of basis points over the reference rate.
Settle	Settlement dates. NINST-by-1 vector of dates representing the settlement dates of the floating rate note.
Maturity	NINST-by-1 vector of dates representing the maturity dates of the floating rate note.
Reset	(Optional) NINST-by-1 vector representing the frequency of payments per year. Default = 1.
Basis	(Optional) NINST-by-1 vector representing the basis used when annualizing the input forward rate tree. Default = 0 (actual/actual).
Principal	(Optional) NINST-by-1 vector of the notional principal amount. Default = 100.
Options	(Optional) Derivatives pricing options structure created with derivset.

Description

[Price, PriceTree] = floatbyhjm(HJMTree, Spread, Settle, Maturity, Reset, Basis, Principal, Options) computes the price of a floating rate note from an HJM tree.

Price is an NINST-by-1 vector of expected prices of the floating rate note at time 0.

PriceTree is a structure of trees containing vectors of instrument prices and accrued interest, and a vector of observation times for each node.

PriceTree.PBush contains the clean prices.

PriceTree.AIBush contains the accrued interest.

PriceTree.tObs contains the observation times.

The Settle date for every floating rate note is set to the ValuationDate of the HJM tree. The floating rate note argument Settle is ignored.

Examples   Price a 20 basis point floating rate note using an HJM forward rate tree.

Load the file deriv.mat, which provides HJMTree. HJMTree contains the time and forward rate information needed to price the note.

```
load deriv
```

Set the required values. Other arguments will use defaults.

```
Spread = 20;
Settle = '01-Jan-2000';
Maturity = '01-Jan-2003';
```

Use floatbyhjm to compute the price of the note.

```
Price = floatbyhjm(HJMTree, Spread, Settle, Maturity)

Price =

  100.5529
```

See Also   bondbyhjm, capbyhjm, cfbyhjm, fixedbyhjm, floorbyhjm, hjmtree, swapbyhjm

Purpose   Price floating rate note prices by a set of zero curves

See Also   Price = floatbyzero(RateSpec, Spread, Settle, Maturity, Reset,
                   Basis, Principal)

Arguments

RateSpec	A structure encapsulating the properties of an interest rate structure. See intenvset for information on creating RateSpec.
Spread	Number of basis points over the reference rate.
Settle	Settlement date. Settle must be earlier than or equal to Maturity.
Maturity	Maturity date.
Reset	(Optional) Frequency of payments per year. Default = 1.
Basis	(Optional) Day count basis. Default = 0 (actual/actual).
Principal	(Optional) The notional principal amount. Default = 100.

All inputs are either scalars or NINST-by-1 vectors unless otherwise specified. Any date may be a serial date number or date string. An optional argument may be passed as an empty matrix [].

Description   Price = floatbyzero(RateSpec, Spread, Settle, Maturity, Reset, Basis, Principal) computes the price of a floating rate note by a set of zero curves.

Price is a number of instruments (NINST) by number of curves (NUMCURVES) matrix of floating rate note prices. Each column arises from one of the zero curves.

Examples   Price a 20 basis point floating rate note using a set of zero curves.

Load the file deriv.mat, which provides ZeroRateSpec, the interest rate term structure needed to price the note.

    load deriv

Set the required values. Other arguments will use defaults.

```
Spread = 20;
Settle = '01-Jan-2000';
Maturity = '01-Jan-2003';
```

Use floatbyzero to compute the price of the note.

```
Price = floatbyzero(ZeroRateSpec, Spread, Settle, Maturity)

Price =

   100.5529
```

See Also   bondbyzero, cfbyzero, fixedbyzero, swapbyzero

Purpose	Price floor instrument by BDT interest rate tree	
Syntax	[Price, PriceTree] = floorbybdt(BDTTree, Strike, Settle, Maturity, Reset, Basis, Principal, Options)	
Arguments	BDTTree	Interest rate tree structure created by bdttree.
	Strike	Number of instruments (NINST)-by-1 vector of rates at which the floor is exercised.
	Settle	Settlement date. NINST-by-1 vector of dates representing the settlement dates of the floor. The Settle date for every floor is set to the ValuationDate of the BDT tree. The floor argument Settle is ignored.
	Maturity	NINST-by-1 vector of dates representing the maturity dates of the floor.
	Reset	(Optional) NINST-by-1 vector representing the frequency of payments per year. Default = 1.
	Basis	(Optional) NINST-by-1 vector representing the basis used when annualizing the input forward rate tree. Default = 0 (actual/actual).
	Principal	(Optional) The notional principal amount. Default = 100.
	Options	(Optional) Derivatives pricing options structure created with derivset.

Description   [Price, PriceTree] = floorbybdt(BDTTree, Strike, Settlement, Maturity, Reset, Basis, Principal, Options) computes the price of a floor instrument from a BDT interest rate tree.

Price is an NINST-by-1 vector of the expected prices of the floor at time 0.

PriceTree is the tree structure with values of the floor at each node.

Examples   Example 1.

Price a 10% floor instrument using a BDT interest rate tree.

Load the file deriv.mat, which provides BDTTree. BDTTree contains the time and interest rate information needed to price the floor instrument.

```
load deriv
```

Set the required values. Other arguments will use defaults.

```
Strike = 0.10;
Settle = '01-Jan-2000';
Maturity = '01-Jan-2004';
```

Use floorbybdt to compute the price of the floor instrument.

```
Price = floorbybdt(BDTTree, Strike, Settle, Maturity)

Price =

    0.1770
```

Example 2.

Here is a second example, showing the pricing of a 10% floor instrument using a newly-created BDT tree.

First set the required arguments for the three needed specifications.

```
Compounding = 1;
ValuationDate = '01-01-2000';
StartDate = ValuationDate;
EndDates = ['01-01-2001'; '01-01-2002'; '01-01-2003';
'01-01-2004'; '01-01-2005'];
Rates = [.1; .11; .12; .125; .13];
Volatility = [.2; .19; .18; .17; .16];
```

Next create the specifications.

```
RateSpec = intenvset('Compounding', Compounding,...
'ValuationDate', ValuationDate,...
'StartDates', StartDate,...
'EndDates', EndDates,...
'Rates', Rates);
BDTTimeSpec = bdttimespec(ValuationDate, EndDates, Compounding);
BDTVolSpec = bdtvolspec(ValuationDate, EndDates, Volatility);
```

Now create the BDT tree from the specifications.

```
BDTTree = bdttree(BDTVolSpec, RateSpec, BDTTimeSpec);
```

Set the floor arguments.

```
FloorStrike = 0.10;
Settlement = ValuationDate;
Maturity = '01-01-2002';
FloorReset = 1;
```

Remaining arguments will use defaults.

Finally, use floorbybdt to find the price of the floor instrument.

```
Price= floorbybdt(BDTTree, FloorStrike, Settlement, Maturity,...
FloorReset)

Price =

    0.0431
```

See Also   bdttree, capbybdt, cfbybdt, swapbybdt

floorbyhjm

Purpose   Price floor instrument by HJM interest rate tree

Syntax
```
[Price, PriceTree] = floorbyhjm(HJMTree, Strike, Settle, Maturity,
    Reset, Basis, Principal, Options)
```

Arguments

HJMTree	Forward rate tree structure created by hjmtree.
Strike	Number of instruments (NINST)-by-1 vector of rates at which the floor is exercised.
Settle	Settlement date. NINST-by-1 vector of dates representing the settlement dates of the floor. The Settle date for every floor is set to the ValuationDate of the HJM tree. The floor argument Settle is ignored.
Maturity	NINST-by-1 vector of dates representing the maturity dates of the floor.
Reset	(Optional) NINST-by-1 vector representing the frequency of payments per year. Default = 1.
Basis	(Optional) NINST-by-1 vector representing the basis used when annualizing the input forward rate tree. Default = 0 (actual/actual).
Principal	(Optional) The notional principal amount. Default = 100.
Options	(Optional) Derivatives pricing options structure created with derivset.

Description   `[Price, PriceTree] = floorbyhjm(HJMTree, Strike, Settlement, Maturity, Reset, Basis, Principal, Options)` computes the price of a floor instrument from an HJM tree.

Price is an NINST-by-1 vector of the expected prices of the floor at time 0.

PriceTree is the tree structure with values of the floor at each node.

Examples   Price a 3% floor instrument using an HJM forward rate tree.

Load the file deriv.mat, which provides HJMTree. HJMTree contains the time and forward rate information needed to price the floor instrument.

```
load deriv
```

Set the required values. Other arguments will use defaults.

```
Strike = 0.03;
Settle = '01-Jan-2000';
Maturity = '01-Jan-2004';
```

Use floorbyhjm to compute the price of the floor instrument.

```
Price = floorbyhjm(HJMTree, Strike, Settle, Maturity)

Price =

  0.0486
```

See Also   capbyhjm, cfbyhjm, hjmtree, swapbyhjm

hedgeopt

Purpose   Allocate optimal hedge for target costs or sensitivities

Syntax   [PortSens, PortCost, PortHolds] = hedgeopt(Sensitivities, Price,
CurrentHolds, FixedInd, NumCosts, TargetCost, TargetSens,
ConSet)

Arguments

Sensitivities	Number of instruments (NINST) by number of sensitivities (NSENS) matrix of dollar sensitivities of each instrument. Each row represents a different instrument. Each column represents a different sensitivity.
Price	NINST-by-1 vector of portfolio instrument unit prices.
CurrentHolds	NINST-by-1 vector of contracts allocated to each instrument.
FixedInd	(Optional) Number of fixed instruments (NFIXED)-by-1 vector of indices of instruments to hold fixed. For example, to hold the first and third instruments of a 10 instrument portfolio unchanged, set FixedInd = [1 3]. Default = [], no instruments held fixed.
NumCosts	(Optional) Number of points generated along the cost frontier when a vector of target costs (TargetCost) is not specified. The default is 10 equally spaced points between the point of minimum cost and the point of minimum exposure. When specifying TargetCost, enter NumCosts as an empty matrix [].
TargetCost	(Optional) Vector of target cost values along the cost frontier. If TargetCost is empty, or not entered, hedgeopt evaluates NumCosts equally spaced target costs between the minimum cost and minimum exposure. When specified, the elements of TargetCost should be positive numbers that represent the maximum amount of money the owner is willing to spend to rebalance the portfolio.

TargetSens	(Optional) 1-by-NSENS vector containing the target sensitivity values of the portfolio. When specifying TargetSens, enter NumCosts and TargetCost as empty matrices [].
ConSet	(Optional) Number of constraints (NCONS) by number of instruments (NINST) matrix of additional conditions on the portfolio reallocations. An eligible NINST-by-1 vector of contract holdings, PortWts, satisfies all the inequalities A*PortWts <= b, where A = ConSet(:,1:end-1) and b = ConSet(:,end).

Notes 1. The user-specified constraints included in ConSet may be created with the functions pcalims or portcons. However, the portcons default PortHolds positivity constraints are typically inappropriate for hedging problems since short-selling is usually required.

2. NPOINTS, the number of rows in PortSens and PortHolds and the length of PortCost, is inferred from the inputs. When the target sensitivities, TargetSens, is entered, NPOINTS = 1; otherwise NPOINTS = NumCosts, or is equal to the length of the TargetCost vector.

3. Not all problems are solvable (e.g., the solution space may be infeasible or unbounded, or the solution may fail to converge). When a valid solution is not found, the corresponding rows of PortSens and PortHolds and the elements of PortCost are padded with NaN's as placeholders.

Description   [PortSens, PortCost, PortHolds] = hedgeopt(Sensitivities, Price, CurrentHolds, FixedInd, NumCosts, TargetCost, TargetSens, ConSet) allocates an optimal hedge by one of two criteria:

- Minimize portfolio sensitivities (exposure) for a given set of target costs
- Minimize the cost of hedging a portfolio given a set of target sensitivities

Hedging involves the fundamental tradeoff between portfolio insurance and the cost of insurance coverage. This function allows investors to modify

portfolio allocations among instruments to achieve either of the criteria. The chosen criterion is inferred from the input argument list. The problem is cast as a constrained linear least-squares problem.

PortSens is a number of points (NPOINTS)-by-NSENS matrix of portfolio sensitivities. When a perfect hedge exists, PortSens is zeros. Otherwise, the best hedge possible is chosen.

PortCost is a 1-by-NPOINTS vector of total portfolio costs.

PortHolds is an NPOINTS-by-NINST matrix of contracts allocated to each instrument. These are the reallocated portfolios.

See Also   hedgeslf

pcalims, portcons, portopt in the *Financial Toolbox User's Guide*

lsqlin in the *Optimization Toolbox User's Guide*

Purpose	Self-financing hedge

Syntax

```
[PortSens, PortValue, PortHolds] = hedgeslf(Sensitivities, Price,
    CurrentHolds, FixedInd, ConSet)
```

Arguments

Sensitivities	Number of instruments (NINST) by number of sensitivities (NSENS) matrix of dollar sensitivities of each instrument. Each row represents a different instrument. Each column represents a different sensitivity.
Price	NINST-by-1 vector of instrument unit prices.
CurrentHolds	NINST-by-1 vector of contracts allocated in each instrument.
FixedInd	(Optional) Empty or number of fixed instruments (NFIXED)-by-1 vector of indices of instruments to hold fixed. The default is FixedInd = 1; the holdings in the first instrument are held fixed. If NFIXED instruments will not be changed, enter all their locations in the portfolio in a vector. If no instruments are to be held fixed, enter FixedInd = [].
ConSet	(Optional) Number of constraints (NCONS)-by-NINST matrix of additional conditions on the portfolio reallocations. An eligible NINST-by-1 vector of contract holdings, PortHolds, satisfies all the inequalities A*PortHolds <= b, where A = ConSet(:,1:end-1) and b = ConSet(:,end).

Description

[PortSens, PortValue, PortHolds] = hedgeslf(Sensitivities, Price, CurrentHolds, FixedInd, ConSet) allocates a self-financing hedge among a collection of instruments. hedgeslf finds the reallocation in a portfolio of financial instruments that hedges the portfolio against market moves and that is closest to being self-financing (maintaining constant portfolio value). By default the first instrument entered is hedged with the other instruments.

PortSens is a 1-by-NSENS vector of portfolio dollar sensitivities. When a perfect hedge exists, PortSens is zeros. Otherwise, the best possible hedge is chosen.

PortValue is the total portfolio value (scalar). When a perfectly self-financing hedge exists, PortValue is equal to dot(Price, CurrentWts) of the initial portfolio.

PortHolds is an NINST-by-1 vector of contracts allocated to each instrument. This is the reallocated portfolio.

Notes 1. The constraints PortHolds(FixedInd) = CurrentHolds(FixedInd) are appended to any constraints passed in ConSet. Pass FixedInd = [] to specify all constraints through ConSet.

2. The default constraints generated by portcons are inappropriate, since they require the sum of all holdings to be positive and equal to one.

3. hedgeself first tries to find the allocations of the portfolio that make it closest to being self-financing, while reducing the sensitivities to 0. If no solution is found, it finds the allocations that minimize the sensitivities. If the resulting portfolio is self-financing, PortValue is equal to the value of the original portfolio.

Examples   Example 1.

Perfect sensitivity cannot be reached.

```
Sens = [0.44  0.32; 1.0 0.0];
Price = [1.2; 1.0];
WO = [1; 1];
[PortSens, PortValue, PortHolds]= hedgeslf(Sens, Price, WO)

PortSens =

    0.0000
    0.3200

PortValue =

    0.7600
```

```
PortHolds =

     1.0000
    -0.4400
```

Example 2.

Constraints are in conflict.

```
Sens = [0.44  0.32; 1.0 0.0];
Price = [1.2; 1.0];
WO = [1; 1];
ConSet = pcalims([2 2])

% O.K. if nothing fixed.

[PortSens, PortValue, PortHolds]= hedgeslf(Sens, Price, WO,...
[], ConSet)

PortSens =

    2.8800
    0.6400

PortValue =

    4.4000

PortHolds =

    2
    2

% WO(1) is not greater than 2.

[PortSens, PortValue, PortHolds] = hedgeslf(Sens, Price, WO,...
1, ConSet)

??? Error using ==> hedgeslf
Overly restrictive allocation constraints implied by ConSet and
by fixing the weight of instruments(s): 1
```

Example 3.

Constraints are impossible to meet.

```
Sens = [0.44  0.32; 1.0 0.0];
Price = [1.2; 1.0];
WO = [1; 1];
ConSet = pcalims([2 2],[1 1]);

[PortSens, PortValue, PortHolds] = hedgeslf(Sens, Price, WO,...
[],ConSet)

??? Error using ==> hedgeslf
Overly restrictive allocation constraints specified in ConSet
```

See Also   hedgeopt

lsqlin in the *Optimization Toolbox User's Guide*

portcons in the *Financial Toolbox User's Guide*

Purpose	Fixed income instrument prices by HJM interest rate tree

Syntax   [Price, PriceTree] = hjmprice(HJMTree, InstSet, Options)

Arguments

HJMTree	Heath-Jarrow-Morton tree sampling a forward rate process. See hjmtree for information on creating HJMTree.
InstSet	Variable containing a collection of instruments. Instruments are categorized by type; each type can have different data fields. The stored data field is a row vector or string for each instrument.
Options	(Optional) Derivatives pricing options structure created with derivset.

Description   Price = hjmprice(HJMTree, InstSet, Options) computes arbitrage free prices for instruments using an interest rate tree created with hjmtree. NINST instruments from a financial instrument variable, InstSet, are priced.

Price is a NINST-by-1 vector of prices for each instrument. The prices are computed by backward dynamic programming on the interest rate tree. If an instrument cannot be priced, NaN is returned.

PriceTree is a MATLAB structure of trees containing vectors of instrument prices and accrued interest, and a vector of observation times for each node.

PriceTree.PBush contains the clean prices.

PriceTree.AIBush contains the accrued interest.

PriceTree.tObs contains the observation times.

hjmprice handles instrument types: 'Bond', 'CashFlow', 'OptBond', 'Fixed', 'Float', 'Cap', 'Floor', 'Swap'. See instadd to construct defined types.

Related single-type pricing functions are:

- bondbyhjm: Price a bond by an HJM tree.
- capbyhjm: Price a cap by an HJM tree.
- cfbyhjm: Price an arbitrary set of cash flows by an HJM tree.
- fixedbyhjm: Price a fixed rate note by an HJM tree.

hjmprice

- `floatbyhjm`: Price a floating rate note by an HJM tree.
- `floorbyhjm`: Price a floor by an HJM tree.
- `optbndbyhjm`: Price a bond option by an HJM tree.
- `swapbyhjm`: Price a swap by an HJM tree.

Examples
Load the HJM tree and instruments from the data file `deriv.mat`. Price the cap and bond instruments contained in the instrument set.

```
load deriv.mat;
HJMSubSet = instselect(HJMInstSet,'Type', {'Bond', 'Cap'});

instdisp(HJMSubSet)
```

```
Index  Type  CouponRate Settle      Maturity    Period Name ...
1      Bond  0.04       01-Jan-2000 01-Jan-2003 1      4% bond
2      Bond  0.04       01-Jan-2000 01-Jan-2004 2      4% bond

Index Type Strike Settle      Maturity    CapReset... Name ...
3     Cap  0.03   01-Jan-2000 01-Jan-2004 1           3% Cap
```

```
[Price, PriceTree] = hjmprice(HJMTree, HJMSubSet)

Warning: Not all cash flows are aligned with the tree. Result will
be approximated.

Price =

   98.7159
   97.5280
    6.2831
PriceTree =

    FinObj: 'HJMPriceTree'
     PBush: {1x5 cell}
    AIBush: {1x5 cell}
      tObs: [0 1 2 3 4]
```

You can use `treeviewer` to see the prices of these three instruments along the price tree.

```
treeviewer(PriceTree, HJMSubSet)
```
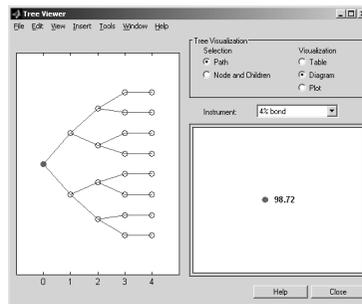

4% Bond (Maturity 2003)
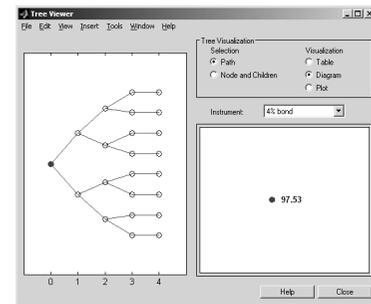
4% Bond (Maturity 2004)
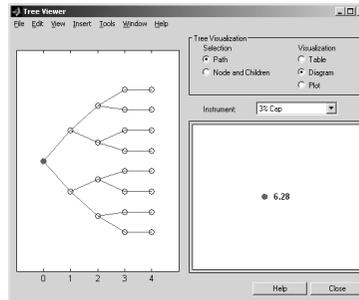
3% Cap

See Also   hjmsens, hjmtree, hjmvolspec, instadd, intenvprice, intenvsens

hjmsens

Purpose   Fixed income instrument prices and sensitivities by HJM interest rate tree

Syntax   `[Delta, Gamma, Vega, Price] = hjmsens(HJMTree, InstSet, Options)`

Arguments

HJMTree	Heath-Jarrow-Morton tree sampling a forward rate process. See `hjmtree` for information on creating HJMTree.
InstSet	Variable containing a collection of instruments. Instruments are categorized by type; each type can have different data fields. The stored data field is a row vector or string for each instrument.
Options	(Optional) Derivatives pricing options structure created with `derivset`.

Description   `[Delta, Gamma, Vega, Price] = hjmsens(HJMTree, InstSet, Options)`
computes instrument sensitivities and prices for instruments using an interest
rate tree created with `hjmtree`. NINST instruments from a financial instrument
variable, InstSet, are priced. `hjmsens` handles instrument types: `'Bond'`,
`'CashFlow'`, `'OptBond'`, `'Fixed'`, `'Float'`, `'Cap'`, `'Floor'`, `'Swap'`. See
`instadd` for information on instrument types.

Delta is an NINST-by-1 vector of deltas, representing the rate of change of
instrument prices with respect to changes in the interest rate. Delta is
computed by finite differences in calls to `hjmtree`. See `hjmtree` for information
on the observed yield curve.

Gamma is an NINST-by-1 vector of gammas, representing the rate of change of
instrument deltas with respect to the changes in the interest rate. Gamma is
computed by finite differences in calls to `hjmtree`.

Vega is an NINST-by-1 vector of vegas, representing the rate of change of
instrument prices with respect to the changes in the volatility $\sigma(t, T)$. Vega is
computed by finite differences in calls to `hjmtree`. See `hjmvolspec` for
information on the volatility process.

Note All sensitivities are returned as dollar sensitivities. To find the per-dollar sensitivities, divide by the respective instrument price.

Price is an NINST-by-1 vector of prices of each instrument. The prices are computed by backward dynamic programming on the interest rate tree. If an instrument cannot be priced, NaN is returned.

Delta and Gamma are calculated based on yield shifts of 100 basis points. Vega is calculated based on a 1% shift in the volatility process.

Examples   Load the tree and instruments from a data file. Compute delta and gamma for the cap and bond instruments contained in the instrument set.

```
load deriv.mat;
HJMSubSet = instselect(HJMInstSet,'Type', {'Bond', 'Cap'});

instdisp(HJMSubSet)
```

```
Index Type   CouponRate Settle      Maturity    Period  Name ...
1     Bond   0.04       01-Jan-2000 01-Jan-2003 1       4% bond
2     Bond   0.04       01-Jan-2000 01-Jan-2004 2       4% bond

Index Type Strike Settle      Maturity    CapReset... Name ...
3     Cap  0.03   01-Jan-2000 01-Jan-2004 1           3% Cap
```

```
  [Delta, Gamma] = hjmsens(HJMTree, HJMSubSet)

  Warning: Not all cash flows are aligned with the tree. Result will
  be approximated.

  Delta =

   -272.6462
   -347.4315
    294.9700
```

```
Gamma =

   1.0e+003 *

      1.0299
      1.6227
      6.8526
```

See Also   hjmprice, hjmtree, hjmvolspec, instadd

Purpose   Specify time structure for HJM interest rate tree

Syntax   TimeSpec = hjmtimespec(ValuationDate, Maturity, Compounding)

Arguments

ValuationDate	Scalar date marking the pricing date and first observation in the tree. Specify as serial date number or date string
Maturity	Number of levels (depth) of the tree. A number of levels (NLEVELS)-by-1 vector of dates marking the cash flow dates of the tree. Cash flows with these maturities fall on tree nodes. Maturity should be in increasing order.
Compounding	(Optional) Scalar value representing the rate at which the input zero rates were compounded when annualized. Default = 1. 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.

Description   TimeSpec = hjmtimespec(ValuationDate, Maturity, Compounding) sets the number of levels and node times for an HJM tree and determines the mapping between dates and time for rate quoting.

TimeSpec is a structure specifying the time layout for hjmtree. The state observation dates are [Settle; Maturity(1:end-1)]. Because a forward rate is stored at the last observation, the tree can value cash flows out to Maturity.

Examples   Specify an eight-period tree with semiannual nodes (every six months). Use exponential compounding to report rates.

```
Compounding = -1;
ValuationDate = '15-Jan-1999';
Maturity = datemnth(ValuationDate, 6*(1:8)');
TimeSpec = hjmtimespec(ValuationDate, Maturity, Compounding)

TimeSpec =

            FinObj: 'HJMTimeSpec'
     ValuationDate: 730135
          Maturity: [8x1 double]
       Compounding: -1
             Basis: 0
       EndMonthRule: 1
```

See Also   hjmtree, hjmvolspec

Purpose   Build an HJM forward rate tree

Syntax   HJMTree = hjmtree(VolSpec, RateSpec, TimeSpec)

Arguments   VolSpec   Volatility process specification. Sets the number of
                                    factors and the rules for computing the volatility
                                    σ(*t*, *T*) for each factor. See hjmvolspec for information on
                                    the volatility process.

                   RateSpec   Interest rate specification for the initial rate curve. See
                                    intenvset for information on declaring an interest rate
                                    variable.

                   TimeSpec   Tree time layout specification. Defines the observation
                                    dates of the HJM tree and the Compounding rule for date
                                    to time mapping and price-yield formulas. See
                                    hjmtimespec for information on the tree structure.

Description   HJMTree = hjmtree(VolSpec, RateSpec, TimeSpec) creates a structure
                   containing time and forward rate information on a bushy tree.

Examples   Using the data provided, create a HJM volatility specification (VolSpec), rate
                   specification (RateSpec), and tree time layout specification (TimeSpec). Then
                   use these specifications to create a HJM tree with hjmtree.

```
Compounding = 1;
ValuationDate = '01-01-2000';
StartDate = ValuationDate;
EndDates = ['01-01-2001'; '01-01-2002'; '01-01-2003';
'01-01-2004'; '01-01-2005'];
Rates = [.1; .11; .12; .125; .13];
Volatility = [.2; .19; .18; .17; .16];
CurveTerm = [ 1; 2; 3; 4; 5];

HJMVolSpec = hjmvolspec('Stationary', Volatility , CurveTerm);
```

```
RateSpec = intenvset('Compounding', Compounding,...
                     'ValuationDate', ValuationDate,...
                     'StartDates', StartDate,...
                     'EndDates', EndDates,...
                     'Rates', Rates);

HJMTimeSpec = hjmtimespec(ValuationDate, EndDates, Compounding);
HJMTree = hjmtree(HJMVolSpec, RateSpec, HJMTimeSpec);
```

Use treeviewer to observe the tree you have created.

```
treeviewer(HJMTree)
```
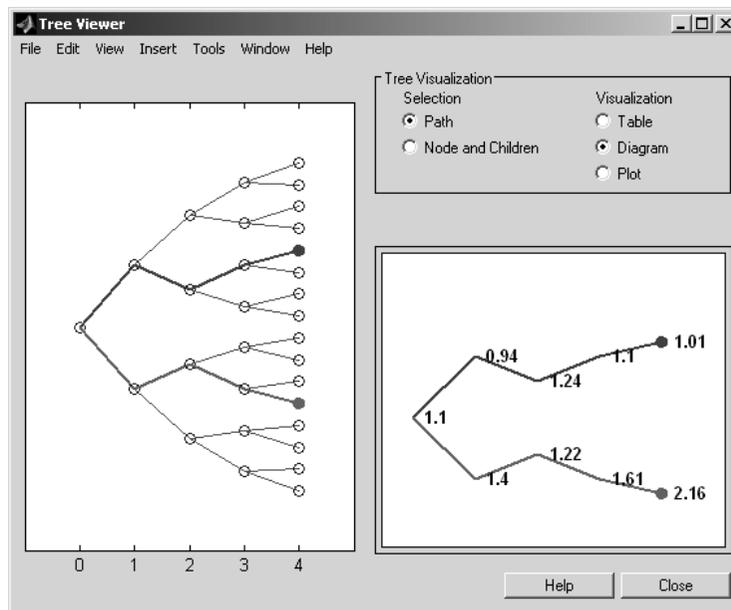

See Also   hjmprice, hjmtimespec, hjmvolspec, intenvset

Purpose   Specify an HJM forward rate volatility process

Syntax   Volspec = hjmvolspec(varargin)

Arguments   The arguments to hjmvolspec vary according to the type and number of volatility factors specified when calling the function. Factors are specified by pairs of names and parameter sets. Factor names can be 'Constant', 'Stationary', 'Exponential', 'Vasicek', or 'Proportional'. The parameter set is specific for each of these factor types:

Constant volatility (Ho-Lee):
VolSpec = hjmvolspec('Constant', Sigma_0)

Stationary volatility:
VolSpec = hjmvolspec('Stationary', CurveVol, CurveTerm)

Exponential volatility:
VolSpec = hjmvolspec('Exponential', Sigma_0, Lambda)

Vasicek, Hull-White:
VolSpec = hjmvolspec('Vasicek', Sigma_0, CurveDecay, CurveTerm)

Nearly proportional stationary:
VolSpec = hjmvolspec('Proportional', CurveProp, CurveTerm, MaxSpot)

You can specify more than one factor by concatenating names and parameter sets.

The table below defines the various arguments to hjmvolspec.

Argument	Description
Sigma_0	Scalar base volatility over a unit time.
Lambda	Scalar decay factor.
CurveVol	Number of curves (NCURVES) -by-1 vector of Vol values at sample points.
CurveDecay	NCURVES-by-1 vector of Decay values at sample points.
CurveProp	NCURVES-by-1 vector of Prop values at sample points.

Argument	Description
CurveTerm	NCURVES-by-1 vector of Term sample points.

Note: See the volatility specifications formulas below for a description of Vol, Decay, Prop, and Term.

Description

Volspec = hjmvolspec(varargin) computes VolSpec, a structure that specifies the volatility model for hjmtree.

hjmvolspec specifies a HJM forward rate volatility process. Each factor is specified with one of the functional forms:

Volatility Specification	Formula
Constant	$\sigma(t, T)$ = Sigma_0
Stationary	$\sigma(t, T)$ = Vol(T-t) = Vol(Term)
Exponential	$\sigma(t, T)$ = Sigma_0*exp(-Lambda*(T-t))
Vasicek, Hull-White	$\sigma(t, T)$ = Sigma_0*exp(-Decay(T-t))
Proportional	$\sigma(t, T)$ = Prop(T-t)*max(SpotRate(t), MaxSpot)

The volatility process is $\sigma(t, T)$, where t is the observation time and T is the starting time of a forward rate. In a stationary process the volatility term is $T - t$. Multiple factors can be specified sequentially.

The time values T, t, and Term are in coupon interval units specified by the Compounding input of hjmtimespec. For instance if Compounding = 2, Term = 1 is a semiannual period (six months).

Examples

Example 1.

Volatility is single-factor proportional.

```
CurveProp = [0.11765; 0.08825; 0.06865];
CurveTerm = [1; 2; 3];
VolSpec = hjmvolspec('Proportional', CurveProp, CurveTerm, 1e6)

 VolSpec =
          FinObj: 'HJMVolSpec'
      FactorModels: {'Proportional'}
```

```
              FactorArgs: {{1x3 cell}}
              SigmaShift: 0
              NumFactors: 1
               NumBranch: 2
                  PBranch: [0.5000 0.5000]
              Fact2Branch: [-1 1]
```

Example 2.

Volatility is two-factor exponential and constant.

```
  VolSpec = hjmvolspec('Exponential', 0.1, 1, 'Constant', 0.2)

  VolSpec =
               FinObj: 'HJMVolSpec'
          FactorModels: {'Exponential'  'Constant'}
            FactorArgs: {{1x2 cell}  {1x1 cell}}
            SigmaShift: 0
            NumFactors: 2
             NumBranch: 3
                PBranch: [0.2500 0.2500 0.5000]
            Fact2Branch: [2x3 double]
```

See Also   hjmtimespec, hjmtree

instadd

Purpose   Add types to instrument collection

Syntax   **Bond instrument.** (See also instbond.)

```
InstSet = instadd('Bond', CouponRate, Settle, Maturity, Period,
    Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate,
    StartDate, Face)
```

Arbitrary cash flow instrument. (See also instcf.)

```
InstSet = instadd('CashFlow', CFlowAmounts, CFlowDates, Settle,
    Basis)
```

Bond option. (See also instoptbnd.)

```
InstSet = instadd('OptBond', BondIndex, OptSpec, Strike,
    ExerciseDates, AmericanOpt)
```

Fixed rate note instrument. (See also instfixed.)

```
InstSet = instadd('Fixed', CouponRate, Settle, Maturity, Reset,
    Basis, Principal) )
```

Floating rate note instrument. (See also instfloat.)

```
InstSet = instadd('Float', Spread, Settle, Maturity, Reset, Basis,
    Principal )
```

Cap instrument. (See also instcap.)

```
InstSet = instadd('Cap', Strike, Settle, Maturity, Reset, Basis,
    Principal)
```

Floor instrument. (See also instfloor.)

```
InstSet = instadd('Floor', Strike, Settle, Maturity, Reset, Basis,
    Principal)
```

Swap instrument. (See also instswap.)

```
InstSet = instadd('Swap', LegRate, Settle, Maturity, LegReset,
    Basis, Principal, LegType)
```

To add instruments to an existing collection:

```
InstSet = instadd(InstSetOld, TypeString, Data1, Data2, ...)
```

Arguments   For more information on instrument data parameters, see the reference entries
                       for individual instrument types. For example, see instcap for additional
                       information on the cap instrument.

InstSetOld	Variable containing a collection of instruments. Instruments are classified by type; each type can have different data fields. The stored data field is a row vector or string for each instrument.

Description   instadd stores instruments of types 'Bond', 'CashFlow', 'OptBond', 'Fixed',
                       'Float', 'Cap', 'Floor', or 'Swap'. Pricing and sensitivity routines are
                       provided for these instruments.

                       InstSet is an instrument set variable containing the new input data.

Examples   Create a portfolio with two cap instruments and a 4% bond.

```
Strike = [0.06; 0.07];
CouponRate = 0.04;
Settle = '06-Feb-2000';
Maturity = '15-Jan-2003';

InstSet = instadd('Cap', Strike, Settle, Maturity);
InstSet = instadd(InstSet, 'Bond', CouponRate, Settle, Maturity);
instdisp(InstSet)
```

Index	Type	Strike	Settle	Maturity	CapReset	Basis	Principal
1	Cap	0.06	06-Feb-2000	15-Jan-2003	NaN	NaN	NaN
2	Cap	0.07	06-Feb-2000	15-Jan-2003	NaN	NaN	NaN

Index	Type	CouponRate	Settle	Maturity ...
3	Bond	0.04	06-Feb-2000	15-Jan-2003...

See Also   instbond, instcap, instcf, instfixed, instfloat, instfloor, instoptbnd,
                       instswap

instaddfield

Purpose   Add new instruments to an instrument collection

Syntax   InstSet = instaddfield('FieldName', FieldList,'Data', DataList,
      'Type',TypeString)
InstSet = instaddfield('FieldName', FieldList, 'FieldClass',
     ClassList, 'Data', DataList, 'Type',TypeString)
InstSetNew = instaddfield(InstSet,'FieldName', FieldList, 'Data',
     DataList, 'Type',TypeString)

Arguments

FieldList	String or number of fields (NFIELDS)-by-1 cell array of strings listing the name of each data field. FieldList cannot be named with the reserved names Type or Index.
DataList	Number of instruments (NINST)-by-M array or NFIELDS-by-1 cell array of data contents for each field. Each row in a data array corresponds to a separate instrument. Single rows are copied to apply to all instruments to be worked on. The number of columns is arbitrary, and data is padded along columns.
ClassList	(Optional) String or NFIELDS-by-1 cell array of strings listing the data class of each field. The class determines how DataList is parsed. Valid strings are 'dble', 'date', and 'char'. The 'FieldClass', ClassList pair is always optional. ClassList is inferred from existing fieldnames or from the data if not entered.
TypeString	String specifying the type of instrument added. Instruments of different types can have different fieldname collections.
InstSet	Variable containing a collection of instruments. Instruments are classified by type; each type can have different data fields. The stored data field is a row vector or string for each instrument.

Description   Use `instaddfield` to create your own types of instruments or to append new instruments to an existing collection. Argument value pairs can be entered in any order.

`InstSet = instaddfield('FieldName', FieldList, 'Data', DataList, 'Type', TypeString)` and

`InstSet = instaddfield('FieldName', FieldList, 'FieldClass', ClassList, 'Data', DataList, 'Type', TypeString)` create an instrument variable.

`InstSetNew = instaddfield(InstSet, 'FieldName', FieldList, 'Data', DataList,'Type ,TypeString)` adds instruments to an existing instrument set, `InstSet`. The output `InstSetNew` is a new instrument set containing the input data.

Examples   Build a portfolio around July options.

```
Strike  Call     Put
 95     12.2     2.9
100      9.2     4.9
105      6.8     7.4

Strike = (95:5:105)'
CallP = [12.2; 9.2; 6.8]
```

Enter three call options with data fields `Strike`, `Price`, and `Opt`.

```
InstSet = instaddfield('Type','Option','FieldName',...
{'Strike','Price','Opt'}, 'Data',{ Strike, CallP, 'Call'});
 instdisp(InstSet)

Index Type    Strike Price Opt
1     Option  95     12.2  Call
2     Option  100     9.2  Call
3     Option  105     6.8  Call
```

Add a futures contract and set the input parsing class.

```
InstSet = instaddfield(InstSet,'Type','Futures',...
'FieldName',{'Delivery','F'},'FieldClass',{'date','dble'},...
'Data' ,{'01-Jul-99',104.4 });
instdisp(InstSet)
```

```
Index Type    Strike Price Opt
1     Option  95    12.2  Call
2     Option 100     9.2  Call
3     Option 105     6.8  Call

Index Type    Delivery      F
4     Futures 01-Jul-1999    104.4
```

Add a put option.

```
FN = instfields(InstSet,'Type','Option')
InstSet = instaddfield(InstSet,'Type','Option',...
'FieldName',FN, 'Data',{105, 7.4, 'Put'});
instdisp(InstSet)

Index Type    Strike Price Opt
1     Option  95    12.2  Call
2     Option 100     9.2  Call
3     Option 105     6.8  Call

Index Type    Delivery      F
4     Futures 01-Jul-1999    104.4

Index Type    Strike Price Opt
5     Option 105     7.4  Put
```

Make a placeholder for another put.

```
InstSet = instaddfield(InstSet,'Type','Option',...
'FieldName','Opt','Data','Put')
instdisp(InstSet)

Index Type    Strike Price Opt
1     Option  95    12.2  Call
2     Option 100     9.2  Call
3     Option 105     6.8  Call

Index Type    Delivery      F
4     Futures 01-Jul-1999    104.4
```

```
Index Type    Strike Price Opt
5     Option 105    7.4  Put
6     Option NaN    NaN  Put
```

Add a cash instrument.

```
InstSet = instaddfield(InstSet, 'Type', 'TBill',...
'FieldName','Price','Data',99)
instdisp(InstSet)

Index Type    Strike Price Opt
1     Option  95   12.2  Call
2     Option 100    9.2  Call
3     Option 105    6.8  Call

Index Type    Delivery      F
4     Futures 01-Jul-1999   104.4

Index Type    Strike Price Opt
5     Option 105    7.4  Put
6     Option NaN    NaN  Put

Index Type  Price
7     TBill 99
```

See Also   instdisp, instget, instgetcell, instsetfield

instbond

Purpose	Construct bond instrument

Syntax	`InstSet = instbond(InstSet, CouponRate, Settle, Maturity, Period,`
	`    Basis,EndMonthRule,IssueDate,FirstCouponDate,LastCouponDate,`
	`    StartDate, Face)`
	`[FieldList, ClassList, TypeString] = instbond`

Arguments

InstSet	Instrument variable. This argument is specified only when adding bond instruments to an existing instrument set. See `instget` for more information on the `InstSet` variable.
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 1, 2, 3, 4, 6, and 12. Default = 2.
Basis	(Optional) Day-count basis of the bond. A vector of integers. 0 = actual/actual (default), 1 = 30/360, 2 = actual/360, 3 = actual/365.
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 is followed only by the bond's maturity cash flow date.
StartDate	Ignored.
Face	(Optional) Face or par value. Default = 100.

Data arguments are number of instruments (NINST)-by-1 vectors, scalar, or empty. Fill unspecified entries in vectors with NaN. Only one data argument is required to create the instrument. The others may be omitted or passed as empty matrices [].

Description   InstSet = instbond(InstSet, CouponRate, Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate, StartDate, Face) creates a new instrument set containing bond instruments or adds bond instruments to a existing instrument set.

[FieldList, ClassList, TypeString] = instbond displays the classes.

FieldList is a number of fields (NFIELDS)-by-1 cell array of strings listing the name of each data field for this instrument type.

ClassList is an NFIELDS-by-1 cell array of strings listing the data class of each field. The class determines how arguments are parsed. Valid strings are 'dble', 'date', and 'char'.

TypeString is a string specifying the type of instrument added. For a bond instrument, TypeString = 'Bond'.

See Also   hjmprice, instaddfield, instdisp, instget, intenvprice

instcap

Purpose	Construct cap instrument
Syntax	InstSet = instcap(InstSet, Strike, Settle, Maturity, Reset, Basis, Principal) [FieldList, ClassList, TypeString] = instcap

Arguments

InstSet	Instrument variable. This argument is specified only when adding cap instruments to an existing instrument set. See instget for more information on the InstSet variable.
Strike	Rate at which the cap is exercised, as a decimal number.
Settle	Settlement date. Serial date number representing the settlement date of the cap.
Maturity	Serial date number representing the maturity date of the cap.
Reset	(Optional) NINST-by-1 vector representing the frequency of payments per year. Default = 1.
Basis	(Optional) NINST-by-1 vector representing the basis used when annualizing the input forward rate tree. Default = 0 (actual/actual).
Principal	(Optional) The notional principal amount. Default = 100.

Description InstSet = instcap(InstSet, Strike, Settle, Maturity, Reset, Basis, Principal) creates a new instrument set containing cap instruments or adds cap instruments to an existing instrument set.

[FieldList, ClassList, TypeString] = instcap displays the classes.

FieldList is a number of fields (NFIELDS)-by-1 cell array of strings listing the name of each data field for this instrument type.

ClassList is an NFIELDS-by-1 cell array of strings listing the data class of each field. The class determines how arguments are parsed. Valid strings are 'dble', 'date', and 'char'.

TypeString is a string specifying the type of instrument added. For a cap instrument, TypeString = 'Cap'.

See Also   `hjmprice, instaddfield, instbond, instdisp, instfloor, instswap, intenvprice`

instcf

Purpose	Construct cash flow instrument

Syntax

```
InstSet = instcf(InstSet, CFlowAmounts, CFlowDates, Settle, Basis)
[FieldList, ClassList, TypeString] = instcf
```

Arguments

InstSet	Instrument variable. This argument is specified only when adding cash flow instruments to an existing instrument set. See instget for more information on the InstSet variable.
CFlowAmounts	Number of instruments (NINST) by maximum number of cash flows (MOSTCFS) matrix of cash flow amounts. Each row is a list of cash flow values for one instrument. If an instrument has fewer than MOSTCFS cash flows, the end of the row is padded with NaNs.
CFlowDates	NINST-by-MOSTCFS matrix of cash flow dates. Each entry contains the date of the corresponding cash flow in CFlowAmounts.
Settle	Settlement date on which the cash flows are priced.
Basis	(Optional) Day-count basis of the bond. A vector of integers. 0 = actual/actual (default), 1 = 30/360, 2 = actual/360, 3 = actual/365.

Only one data argument is required to create an instrument. Other arguments can be omitted or passed as empty matrices []. Dates can be input as serial date numbers or date strings.

Description

`InstSet = instcf(InstSet, CFlowAmounts, CFlowDates, Settle, Basis)` creates a new instrument set from data arrays or adds instruments of type CashFlow to an instrument set.

`[FieldList, ClassList, TypeString] = instcf` lists field meta-data for an instrument of type CashFlow.

FieldList is a number of fields (NFIELDS)-by-1 cell array of strings listing the name of each data field for this instrument type.

ClassList is an NFIELDS-by-1 cell array of strings listing the data class of each field. The class determines how arguments are parsed. Valid strings are `'dble'`, `'date'`, and `'char'`.

TypeString specifies the type of instrument added, e.g., TypeString = `'CashFlow'`.

See Also   hjmprice, instaddfield, instdisp, instget, intenvprice

instdelete

Purpose	Complement of a subset of instruments found by matching conditions

Syntax

```
ISubSet = instdelete(InstSet, 'FieldName', FieldList, 'Data',
   DataList, 'Index', IndexSet, 'Type', TypeList)
```

Arguments

InstSet	Variable containing a collection of instruments. Instruments are classified by type; each type can have different data fields. The stored data field is a row vector or string for each instrument.
FieldList	String or number of fields (NFIELDS)-by-1 cell array of strings listing the name of each data field to match with data values.
DataList	Number of values (NVALUES)-by-M array or NFIELDS-by-1 cell array of acceptable data values for each field. Each row lists a data row value to search for in the corresponding FieldList. The number of columns is arbitrary and matching will ignore trailing NaNs or spaces.
IndexSet	(Optional) Number of instruments (NINST)-by-1 vector restricting positions of instruments to check for matches. The default is all indices available in the instrument variable.
TypeList	(Optional) String or number of types (NTYPES)-by-1 cell array of strings restricting instruments to match one of TypeList types. The default is all types in the instrument variable.

Argument value pairs can be entered in any order. The InstSet variable must be the first argument. 'FieldName' and 'Data' arguments must appear together or not at all.

Description

The output argument ISubSet contains instruments *not* matching the input criteria. Instruments are deleted from ISubSet if all the Field, Index, and Type conditions are met. An instrument meets an individual Field condition if the stored FieldName data matches any of the rows listed in the DataList for that FieldName. See instfind for more examples on matching criteria.

Examples   Retrieve the instrument set variable ExampleInst from the data file.
InstSetExamples.mat. The variable contains three types of instruments:
Option, Futures, and TBill.

```
load InstSetExamples;
instdisp(ExampleInst)

Index Type    Strike Price Opt  Contracts
1     Option 95    12.2  Call    0
2     Option 100    9.2  Call    0
3     Option 105    6.8  Call  1000

Index Type    Delivery       F     Contracts
4     Futures 01-Jul-1999   104.4 -1000

Index Type    Strike Price Opt  Contracts
5     Option 105    7.4  Put  -1000
6     Option 95     2.9  Put     0

Index Type  Price  Maturity      Contracts
7     TBill 99     01-Jul-1999   6
```

Create a new variable, ISet, with all options deleted.

```
ISet = instdelete(ExampleInst, 'Type','Option');
instdisp(ISet)

Index Type    Delivery       F     Contracts
1     Futures 01-Jul-1999   104.4 -1000

Index Type  Price  Maturity      Contracts
2     TBill 99     01-Jul-1999   6
```

See Also   instaddfield, instfind, instget, instselect

instdisp

Purpose   Display instruments

Syntax   CharTable = instdisp(InstSet)

Arguments   InstSet   Variable containing a collection of instruments. See
                                 instaddfield for examples on constructing the variable.

Description CharTable = instdisp(InstSet) creates a character array displaying the
                contents of an instrument collection, InstSet. If instdisp is called without
                output arguments, the table is displayed in the command window.

                CharTable is a character array with a table of instruments in InstSet. For
                each instrument row, the Index and Type are printed along with the field
                contents. Field headers are printed at the tops of the columns.

Examples   Retrieve the instrument set ExampleInst from the data file.
                InstSetExamples.mat. ExampleInst contains three types of instruments:
                Option, Futures, and TBill.

```
load InstSetExamples;
instdisp(ExampleInst)

Index Type   Strike Price Opt  Contracts
1     Option 95     12.2 Call     0
2     Option 100     9.2 Call     0
3     Option 105     6.8 Call  1000

Index Type    Delivery      F      Contracts
4     Futures 01-Jul-1999   104.4 -1000

Index Type   Strike Price Opt  Contracts
5     Option 105     7.4 Put  -1000
6     Option  95     2.9 Put      0

Index Type Price Maturity      Contracts
7     TBill 99    01-Jul-1999   6
```

See Also   datestr in the *Financial Toolbox User's Guide*

num2str in the online MATLAB Reference

instaddfield, instget

instfields

Purpose	List fields
Syntax	FieldList = instfields(InstSet, 'Type', TypeList)

Arguments

InstSet	Variable containing a collection of instruments. Instruments are classified by type; each type can have different data fields. The stored data field is a row vector or string for each instrument.
TypeList	(Optional) String or number of types (NTYPES)-by-1 cell array of strings listing the instrument types to query.

Description

FieldList = instfields(InstSet, 'Type', TypeList) retrieve list of fields stored in an instrument variable.

FieldList is a number of fields (NFIELDS)-by-1 cell array of strings listing the name of each data field corresponding to the listed types.

Examples

Retrieve the instrument set ExampleInst from the data file. InstSetExamples.mat. ExampleInst contains three types of instruments: Option, Futures, and TBill.

```
load InstSetExamples;
instdisp(ExampleInst)

Index Type    Strike Price Opt  Contracts
1     Option  95    12.2   Call    0
2     Option 100     9.2   Call    0
3     Option 105     6.8   Call  1000

Index Type    Delivery      F       Contracts
4     Futures 01-Jul-1999  104.4   -1000

Index Type    Strike Price Opt  Contracts
5     Option 105      7.4 Put  -1000
6     Option  95      2.9 Put     0

Index Type Price Maturity       Contracts
7     TBill 99    01-Jul-1999    6
```

Get the fields listed for type `'Option'`.

```
[FieldList, ClassList] = instfields(ExampleInst, 'Type',...
'Option')

FieldList =

    'Strike'
    'Price'
    'Opt'
    'Contracts'

ClassList =

    'dble'
    'dble'
    'char'
    'dble'
```

Get the fields listed for types `'Option'` and `'TBill'`.

```
FieldList = instfields(ExampleInst, 'Type', {'Option', 'TBill'})

FieldList =

    'Strike'
    'Opt'
    'Price'
    'Maturity'
    'Contracts'
```

Get all the fields listed in any type in the variable.

```
FieldList = instfields(ExampleInst)
```

instfields

FieldList =

    'Delivery'
    'F'
    'Strike'
    'Opt'
    'Price'
    'Maturity'
    'Contracts'

See Also   instdisp, instlength, insttypes

Purpose	Search instruments for matching conditions
Syntax	IndexMatch = instfind(InstSet, 'FieldName', FieldList, 'Data', DataList, 'Index', IndexSet, 'Type', TypeList)

Arguments

InstSet	Variable containing a collection of instruments. Instruments are classified by type; each type can have different data fields. The stored data field is a row vector or string for each instrument.
FieldList	String or number of fields (NFIELDS)-by-1 cell array of strings listing the name of each data field to match with data values.
DataList	Number of values (NVALUES)-by-M array or NFIELDS-by-1 cell array of acceptable data values for each field. Each row lists a data row value to search for in the corresponding FieldList. The number of columns is arbitrary, and matching will ignore trailing NaNs or spaces.
IndexSet	(Optional) Number of instruments (NINST)-by-1 vector restricting positions of instruments to check for matches. The default is all indices available in the instrument variable.
TypeList	(Optional) String or number of types (NTYPES)-by-1 cell array of strings restricting instruments to match one of TypeList types. The default is all types in the instrument variable.

Argument value pairs can be entered in any order. The InstSet variable must be the first argument. 'FieldName' and 'Data' arguments must appear together or not at all.

Description   IndexMatch = instfind(InstSet, 'FieldName', FieldList, 'Data', DataList,'Index', IndexSet, 'Type', TypeList) returns indices of instruments matching Type, Field, or Index values.

IndexMatch is an NINST-by-1 vector of positions of instruments matching the input criteria. Instruments are returned in IndexMatch if all the Field, Index,

and `Type` conditions are met. An instrument meets an individual `Field` condition if the stored `FieldName` data matches any of the rows listed in the `DataList` for that `FieldName`.

Examples   Retrieve the instrument set `ExampleInst` from the data file. `InstSetExamples.mat`. `ExampleInst` contains three types of instruments: `Option`, `Futures`, and `TBill`.

```
load InstSetExamples;
instdisp(ExampleInst)

Index Type    Strike Price Opt  Contracts
1      Option 95     12.2  Call    0
2      Option 100     9.2  Call    0
3      Option 105     6.8  Call  1000

Index Type    Delivery       F    Contracts
4      Futures 01-Jul-1999   104.4 -1000

Index Type    Strike   Price Opt  Contracts
5      Option 105       7.4  Put  -1000
6      Option  95       2.9  Put     0

Index Type  Price Maturity       Contracts
7      TBill 99     01-Jul-1999     6
```

Make a vector, `Opt95`, containing the indexes within `ExampleInst` of the options struck at 95.

```
Opt95 = instfind(ExampleInst, 'FieldName','Strike','Data', 95 )

Opt95 =

    1
    6
```

Locate the futures and Treasury bill instruments within `ExampleInst`.

```
Types = instfind(ExampleInst,'Type',{'Futures';'TBill'})

Types =

     4
     7
```

See Also   instaddfield, instget, instgetcell, instselect

instfixed

Purpose	Construct fixed-rate instrument
Syntax	InstSet = instfixed(InstSet, CouponRate, Settle, Maturity, Reset, Basis, Principal) [FieldList, ClassList, TypeString] = instfixed

Arguments

InstSet	Instrument variable. This argument is specified only when adding fixed rate note instruments to an existing instrument set. See instget for more information on the InstSet variable.
CouponRate	Decimal annual rate.
Settle	Settlement date. Date string or serial date number representing the settlement date of the fixed rate note.
Maturity	Date string or serial date number representing the maturity date of the fixed rate note.
Reset	(Optional) NINST-by-1 vector representing the frequency of payments per year. Default = 1.
Basis	(Optional) NINST-by-1 vector representing the basis used when annualizing the input forward rate tree. Default = 0 (actual/actual).
Principal	(Optional) The notional principal amount. Default = 100.

Data arguments are number of instruments (NINST)-by-1 vectors, scalar, or empty. Fill unspecified entries in vectors with NaN. Only one data argument is required to create the instrument. The others may be omitted or passed as empty matrices [ ].

Description InstSet = instfixed(InstSet, Strike, Settle, Maturity, Reset, Basis, Principal) creates a new instrument set containing fixed rate instruments or adds fixed rate instruments to an existing instrument set.

[FieldList, ClassList, TypeString] = instfixed displays the classes.

FieldList is a number of fields (NFIELDS)-by-1 cell array of strings listing the name of each data field for this instrument type.

ClassList is an NFIELDS-by-1 cell array of strings listing the data class of each field. The class determines how arguments are parsed. Valid strings are `'dble'`, `'date'`, and `'char'`.

TypeString is a string specifying the type of instrument added. For a fixed rate instrument, TypeString = `'Fixed'`.

See Also   hjmprice, instaddfield, instbond, instcap, instdisp, instswap, intenvprice

instfloat

Purpose	Construct floating-rate instrument

Syntax

```
InstSet = instfloat(InstSet, Spread, Settle, Maturity, Reset, Basis,
  Principal)
[FieldList, ClassList, TypeString] = instfloat
```

Arguments

InstSet	Instrument variable. This argument is specified only when adding floating rate note instruments to an existing instrument set. See instget for more information on the InstSet variable.
Spread	Number of basis points over the reference rate.
Settle	Settlement date. Date string or serial date number representing the settlement date of the floating rate note.
Maturity	Date string or serial date number representing the maturity date of the floating rate note.
Reset	(Optional) NINST-by-1 vector representing the frequency of payments per year. Default = 1.
Basis	(Optional) NINST-by-1 vector representing the basis used when annualizing the input forward rate tree. Default = 0 (actual/actual).
Principal	(Optional) The notional principal amount. Default = 100.

Data arguments are number of instruments (NINST)-by-1 vectors, scalar, or empty. Fill unspecified entries in vectors with NaN. Only one data argument is required to create the instrument. The others may be omitted or passed as empty matrices [].

Description

InstSet = instfloat(InstSet, Spread, Settle, Maturity, Reset, Basis, Principal) creates a new instrument set containing floating rate instruments or adds floating rate instruments to an existing instrument set.

[FieldList, ClassList, TypeString] = instfloat displays the classes.

FieldList is a number of fields (NFIELDS)-by-1 cell array of strings listing the name of each data field for this instrument type.

ClassList is an NFIELDS-by-1 cell array of strings listing the data class of each field. The class determines how arguments are parsed. Valid strings are `'dble'`, `'date'`, and `'char'`.

TypeString is a string specifying the type of instrument added. For a floating rate instrument, TypeString = `'Float'`.

See Also   `hjmprice, instaddfield, instbond, instcap, instdisp, instswap, intenvprice`

instfloor

Purpose	Construct floor instrument

Syntax

```
InstSet = instfloor(InstSet, Strike, Settle, Maturity, Reset, Basis,
    Principal)
[FieldList, ClassList, TypeString] = instfloor
```

Arguments

InstSet	Instrument variable. This argument is specified only when adding floor instruments to an existing instrument set. See instget for more information on the InstSet variable.
Strike	Rate at which the floor is exercised, as a decimal number.
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.
Reset	(Optional) NINST-by-1 vector representing the frequency of payments per year. Default = 1.
Basis	(Optional) Day-count basis of the bond. A vector of integers. 0 = actual/actual (default), 1 = 30/360, 2 = actual/360, 3 = actual/365.
Principal	(Optional) The notional principal amount. Default = 100.

Description

InstSet = instfloor(InstSet, Strike, Settle, Maturity, Reset, Basis, Principal) creates a new instrument set containing floor instruments or adds floor instruments to an existing instrument set.

[FieldList, ClassList, TypeString] = instfloor displays the classes.

FieldList is a number of fields (NFIELDS)-by-1 cell array of strings listing the name of each data field for this instrument type.

ClassList is an NFIELDS-by-1 cell array of strings listing the data class of each field. The class determines how arguments are parsed. Valid strings are `'dble'`, `'date'`, and `'char'`.

TypeString is a string specifying the type of instrument added. For a floor instrument, TypeString = `'Floor'`.

See Also   hjmprice, instaddfield, instbond, instcap, instdisp, instswap, intenvprice

instget

Purpose	Retrieve data from instrument variable
Syntax	`[Data_1, Data_2,...,Data_n] = instget(InstSet, 'FieldName',` `FieldList, 'Index', IndexSet, 'Type', TypeList)`

Arguments

`InstSet`	Variable containing a collection of instruments. Instruments are classified by type; each type can have different data fields. The stored data field is a row vector or string for each instrument.
`FieldList`	(Optional) String or number of fields (NFIELDS)-by-1 cell array of strings listing the name of each data field to match with data values. `FieldList` entries can also be either `'Type'` or `'Index'`; these return type strings and index numbers respectively. The default is all fields available for the returned set of instruments.
`IndexSet`	(Optional) Number of instruments (NINST)-by-1 vector of positions of instruments to work on. If `TypeList` is also entered, instruments referenced must be one of `TypeList` types and contained in `IndexSet`. The default is all indices available in the instrument variable.
`TypeList`	(Optional) String or number of types (NTYPES)-by-1 cell array of strings restricting instruments to match one of `TypeList` types. The default is all types in the instrument variable.

Parameter value pairs can be entered in any order. The `InstSet` variable must be the first argument.

Description   `[Data_1, Data_2,...,Data_n] = instget(InstSet, 'FieldName',`
`FieldList, 'Index', IndexSet, 'Type', TypeList)` retrieve data arrays from an instrument variable.

`Data_1` is an NINST-by-M array of data contents for the first field in `FieldList`. Each row corresponds to a separate instrument in `IndexSet`. Unavailable data is returned as NaN or as spaces.

`Data_n` is an NINST-by-M array of data contents for the last field in `FieldList`.

Examples   Retrieve the instrument set ExampleInst from the data file.
InstSetExamples.mat. ExampleInst contains three types of instruments:
Option, Futures, and TBill.

```
load InstSetExamples;
instdisp(ExampleInst)

Index Type    Strike Price Opt  Contracts
1     Option  95    12.2  Call     0
2     Option  100    9.2  Call     0
3     Option  105    6.8  Call  1000

Index Type    Delivery       F     Contracts
4     Futures 01-Jul-1999   104.4 -1000

Index Type    Strike Price Opt  Contracts
5     Option  105    7.4  Put  -1000
6     Option  95     2.9  Put      0

Index Type Price Maturity       Contracts
7     TBill 99    01-Jul-1999  6
```

Extract the price from all instruments.

```
P = instget(ExampleInst,'FieldName','Price')

P =

   12.2000
    9.2000
    6.8000
       NaN
    7.4000
    2.9000
   99.0000
```

Get all the prices and the number of contracts held.

```
[P,C] = instget(ExampleInst, 'FieldName', {'Price', 'Contracts'})
```

```
P =

   12.2000
    9.2000
    6.8000
       Nan
    7.4000
    2.9000
   99.0000


C =

        0
        0
     1000
    -1000
    -1000
        0
        6
```

Compute a value V. Create a new variable ISet that appends V to ExampleInst.

```
V = P.*C
ISet = instsetfield(ExampleInst, 'FieldName', 'Value', 'Data',...
V);
instdisp(ISet)

Index Type    Strike Price Opt  Contracts Value
1     Option  95      12.2 Call    0           0
2     Option 100       9.2 Call    0           0
3     Option 105       6.8 Call 1000        6800

Index Type    Delivery        F      Contracts Value
4     Futures 01-Jul-1999  104.4 -1000     NaN

Index Type    Strike Price Opt  Contracts Value
5     Option 105       7.4 Put  -1000      -7400
6     Option  95       2.9 Put     0           0

Index Type Price Maturity        Contracts Value
7     TBill 99    01-Jul-1999     6         594
```

Look at only the instruments which have nonzero Contracts.

```
Ind = find(C ~= 0)

Ind =

     3
     4
     5
     7
```

Get the Type and Opt parameters from those instruments. (Only options have a stored 'Opt' field.)

```
[T,O] = instget(ExampleInst, 'Index', Ind, 'FieldName',...
{'Type', 'Opt'})

T =

Option
Futures
Option
TBill

O =

Call

Put
```

Create a string report of holdings Type, Opt, and Value.

```
rstring = [T, O, num2str(V(Ind))]

rstring =

Option Call    6800
Futures         NaN
Option Put    -7400
TBill           594
```

See Also   instaddfield, instdisp, instgetcell

instgetcell

Purpose   Retrieve data and context from instrument variable

Syntax
```
[DataList, FieldList, ClassList, IndexSet, TypeSet] =
    instgetcell(InstSet, 'FieldName', FieldList, 'Index', IndexSet,
    'Type', TypeList)
```

Arguments

InstSet	Variable containing a collection of instruments. Instruments are classified by type; each type can have different data fields. The stored data field is a row vector or string for each instrument.
FieldList	(Optional) String or number of fields (NFIELDS)-by-1 cell array of strings listing the name of each data field to match with data values. FieldList should not be either Type or Index; these field names are reserved. The default is all fields available for the returned set of instruments.
IndexSet	(Optional) Number of instruments (NINST)-by-1 vector of positions of instruments to work on. If TypeList is also entered, instruments referenced must be one of TypeList types and contained in IndexSet. The default is all indices available in the instrument variable.
TypeList	(Optional) String or number of types (NTYPES)-by-1 cell array of strings restricting instruments to match one of TypeList types. The default is all types in the instrument variable.

Parameter value pairs can be entered in any order. The InstSet variable must be the first argument.

Description   `[DataList, FieldList, ClassList] = instgetcell(InstSet, 'FieldName', FieldList, 'Index', IndexSet, 'Type', TypeList)` retrieves data and context from an instrument variable.

DataList is an NFIELDS-by-1 cell array of data contents for each field. Each cell is an NINST-by-M array, where each row corresponds to a separate instrument in IndexSet. Any data which is not available is returned as NaN or as spaces.

FieldList is an NFIELDS-by-1 cell array of strings listing the name of each field in DataList.

ClassList is an NFIELDS-by-1 cell array of strings listing the data class of each field. The class determines how arguments are parsed. Valid strings are `'dble'`, `'date'`, and `'char'`.

IndexSet is an NINST-by-1 vector of positions of instruments returned in DataList.

TypeSet is an NINST-by-1 cell array of strings listing the type of each instrument row returned in DataList.

Examples   Retrieve the instrument set ExampleInst from the data file InstSetExamples.mat. ExampleInst contains three types of instruments: Option, Futures, and TBill.

```
load InstSetExamples;
instdisp(ExampleInst)

Index Type    Strike Price Opt  Contracts
1     Option  95     12.2  Call    0
2     Option 100      9.2  Call    0
3     Option 105      6.8  Call  1000

Index Type    Delivery        F    Contracts
4     Futures 01-Jul-1999   104.4 -1000

Index Type    Strike Price Opt  Contracts
5     Option 105      7.4 Put  -1000
6     Option  95      2.9 Put     0

Index Type Price Maturity       Contracts
7     TBill 99    01-Jul-1999    6
```

Get the prices and contracts from all instruments.

```
FieldList = {'Price'; 'Contracts'}
DataList = instgetcell(ExampleInst, 'FieldName', FieldList )
P = DataList{1}
C = DataList{2}
```

```
P =

    12.2000
     9.2000
     6.8000
         NaN
     7.4000
     2.9000
    99.0000


C =

        0
        0
     1000
    -1000
    -1000
        0
        6
```

Get all the option data: Strike, Price, Opt, Contracts.

```
[DataList, FieldList, ClassList] = instgetcell(ExampleInst,...
'Type','Option')

DataList =

    [5x1 double]
    [5x1 double]
    [5x4 char  ]
    [5x1 double]


FieldList =

    'Strike'
    'Price'
    'Opt'
    'Contracts'
```

```
ClassList =

    'dble'
    'dble'
    'char'
    'dble'
```

Look at the data as a comma separated list. Type help lists for more information on cell array lists.

```
DataList{:}

ans =

     95
    100
    105
    105
     95

ans =

    12.2100
     9.2000
     6.8000
     7.3900
     2.9000

ans =

    Call
    Call
    Call

    Put
    Put
```

```
ans =

        0
        0
      100
     -100
        0
```

See Also   instaddfield, instdisp, instget

Purpose   Count instruments

Syntax   NInst = instlength(InstSet)

Arguments

InstSet	Variable containing a collection of instruments. Instruments are classified by type; each type can have different data fields. The stored data field is a row vector or string for each instrument.

Description   NInst = instlength(InstSet) computes NInst, the number of instruments contained in the variable, InstSet.

See Also   instdisp, instfields, insttypes

instoptbnd

Purpose	Construct bond option
Syntax	InstSet = instoptbnd(InstSet, BondIndex, OptSpec, Strike, ExerciseDates, AmericanOpt) [FieldList, ClassList, TypeString] = instoptbnd

Arguments

InstSet	Variable containing a collection of instruments. Instruments are classified by type; each type can have different data fields. The stored data field is a row vector or string for each instrument.
BondIndex	Number of instruments (NINST)-by-1 vector of indices pointing to underlying instruments of Type 'Bond' which are also stored in InstSet. See instbond for information on specifying the bond data.
OptSpec	NINST-by-1 list of string values 'Call' or 'Put'.

The interpretation of the Strike and ExerciseDates arguments depends upon the setting of the AmericanOpt argument. If AmericanOpt = 0, NaN, or is unspecified, the option is a European or Bermuda option. If AmericanOpt = 1, the option is an American option.

Strike	For a European or Bermuda option: NINST by number of strikes (NSTRIKES) matrix of strike price values. Each row is the schedule for one option. If an option has fewer than NSTRIKES exercise opportunities, the end of the row is padded with NaNs. For an American option: NINST-by-1 vector of strike price values for each option.

ExerciseDates	For a European or Bermuda option: NINST-by-NSTRIKES matrix of exercise dates. Each row is the schedule for one option. For a European option, there is only one exercise date, the option expiry date.
	For an American option: NINST-by-2 vector of exercise date boundaries. For each instrument, the option can be exercised on any coupon date between or including the pair of dates on that row. If only one non-NaN date is listed, or if ExerciseDates is NINST-by-1, the option can be exercised between the underlying bond Settle and the single listed exercise date.

Data arguments are NINST-by-1 vectors, scalar, or empty. Fill unspecified entries in vectors with NaN. Only one data argument is required to create the instrument. The others may be omitted or passed as empty matrices [].

Description   InstSet = instoptbnd(InstSet, BondIndex, OptSpec, Strike, ExerciseDates) specifies a European or Bermuda option.

InstSet = instoptbnd(InstSet, BondIndex, OptSpec, Strike, ExerciseDates, AmericanOpt) specifies an American option if AmericanOpt is set to 1. If AmericanOpt is not set to 1, the function specifies a European or Bermuda option.

FieldList is a number of fields (NFIELDS)-by-1 cell array of strings listing the name of each data field for this instrument type.

ClassList is an NFIELDS-by-1 cell array of strings listing the data class of each field. The class determines how arguments are parsed. Valid strings are 'dble', 'date', and 'char'.

TypeString is a string specifying the type of instrument added. For a bond instrument, TypeString = 'Bond'.

See Also   hjmprice, instadd, instdisp, instget

instselect

Purpose   Create instrument subset by matching conditions

Syntax
```
InstSubSet = instselect(InstSet, 'FieldName', FieldList, 'Data',
   DataList,    'Index', IndexSet, 'Type', TypeList)
```

Arguments

InstSet	Variable containing a collection of instruments. Instruments are classified by type; each type can have different data fields. The stored data field is a row vector or string for each instrument.
FieldList	String or number of fields (NFIELDS)-by-1 cell array of strings listing the name of each data field to match with data values.
DataList	Number of values (NVALUES)-by-M array or NFIELDS-by-1 cell array of acceptable data values for each field. Each row lists a data row value to search for in the corresponding FieldList. The number of columns is arbitrary and matching will ignore trailing NaNs or spaces.
IndexSet	(Optional) Number of instruments (NINST)-by-1 vector restricting positions of instruments to check for matches. The default is all indices available in the instrument variable.
TypeList	(Optional) String or number of types (NTYPES)-by-1 cell array of strings restricting instruments to match one of TypeList types. The default is all types in the instrument variable.

Parameter value pairs can be entered in any order. The InstSet variable must be the first argument. 'FieldName' and 'Data' parameters must appear together or not at all. 'Index' and 'Type' parameters are each optional.

Description
```
InstSubSet = instselect(InstSet, 'FieldName', FieldList, 'Data',
DataList,    'Index', IndexSet, 'Type', TypeList)
```
creates an instrument subset (InstSubSet) from an existing set of instruments (InstSet).

InstSubSet is a variable containing instruments matching the input criteria. Instruments are returned in InstSubSet if all the Field, Index, and Type

conditions are met. An instrument meets an individual Field condition if the stored FieldName data matches any of the rows listed in the DataList for that FieldName. See instfind for examples on matching criteria.

Examples
Retrieve the instrument set ExampleInst from the data file. InstSetExamples.mat. The variable contains three types of instruments: Option, Futures, and TBill.

```
load InstSetExamples
instdisp(ExampleInst)

Index Type    Strike Price Opt  Contracts
1     Option  95    12.2  Call    0
2     Option 100     9.2  Call    0
3     Option 105     6.8  Call  1000

Index Type    Delivery       F     Contracts
4     Futures 01-Jul-1999   104.4 -1000

Index Type   Strike   Price Opt  Contracts
5     Option 105      7.4  Put  -1000
6     Option 95       2.9  Put     0

Index Type  Price Maturity       Contracts
7     TBill 99    01-Jul-1999     6
```

Make a new portfolio containing only options struck at 95.

```
Opt95 = instselect(ExampleInst, 'FieldName', 'Strike',...
'Data', '95')

instdisp(Opt95)

Opt95 =

Index Type    Strike Price Opt  Contracts
1     Option  95    12.2  Call    0
2     Option  95     2.9  Put     0
```

Make a new portfolio containing only futures and Treasury bills.

```
FutTBill = instselect(ExampleInst,'Type',{'Futures';'TBill'})

instdisp(FutTBill) =

Index Type    Delivery       F     Contracts
1     Futures 01-Jul-1999    104.4 -1000

Index Type  Price Maturity      Contracts
2     TBill 99    01-Jul-1999   6
```

See Also   instaddfield, instdelete, instfind, instget, instgetcell

Purpose   Add or reset data for existing instruments

Syntax   InstSet = instsetfield(InstSet, 'FieldName', FieldList, 'Data',
                  DataList)
               InstSet = instsetfield(InstSet, 'FieldName', FieldList, 'Data',
                  DataList, 'Index', IndexSet, 'Type', TypeList)

Arguments

InstSet	(Required) Variable containing a collection of instruments. Instruments are classified by type; each type can have different data fields. The stored data field is a row vector or string for each instrument. InstSet must be the first argument in the list.
FieldList	String or number of fields (NFIELDS)-by-1 cell array of strings listing the name of each data field. FieldList cannot be named with the reserved names Type or Index.
DataList	Number of instruments (NINST)-by-M array or NFIELDS-by-1 cell array of data contents for each field. Each row in a data array corresponds to a separate instrument. Single rows are copied to apply to all instruments to be worked on. The number of columns is arbitrary, and data is padded along columns.
IndexSet	NINST-by-1 vector of positions of instruments to work on. If TypeList is also entered, instruments referenced must be one of TypeList types and contained in IndexSet.
TypeList	String or number of types (NTYPES)-by-1 cell array of strings restricting instruments worked on to match one of TypeList types.

Argument value pairs can be entered in any order.

Description   instsetfield sets data for existing instruments in a collection variable.

InstSet = instsetfield(InstSet, 'FieldName', FieldList, 'Data',
DataList) resets or adds fields to every instrument.

instsetfield

InstSet = instsetfield(InstSet, 'FieldName', FieldList, 'Data', DataList, 'Index', IndexSet, 'Type', TypeList) resets or adds fields to a subset of instruments.

The output InstSet is a new instrument set variable containing the input data.

Examples

Retrieve the instrument set ExampleInstSF from the data file. InstSetExamples.mat. ExampleInstSF contains three types of instruments: Option, Futures, and TBill.

```
load InstSetExamples;
ISet = ExampleInstSF;
instdisp(ISet)

Index Type    Strike Price Opt
1     Option  95     12.2 Call
2     Option 100      9.2 Call
3     Option 105      6.8 Call

Index Type     Delivery       F
4     Futures 01-Jul-1999    104.4

Index Type    Strike Price Opt
5     Option 105      7.4 Put
6     Option NaN      NaN  Put

Index Type  Price
7     TBill 99
```

Enter data for the option in Index 6: Price 2.9 for a Strike of 95.

```
ISet = instsetfield(ISet, 'Index',6,...
'FieldName',{'Strike','Price'}, 'Data',{ 95 , 2.9 });
instdisp(ISet)

Index Type    Strike Price Opt
1     Option  95     12.2 Call
2     Option 100      9.2 Call
3     Option 105      6.8 Call
Index Type     Delivery       F
4     Futures 01-Jul-1999    104.4
```

```
Index Type    Strike Price Opt
5     Option 105    7.4  Put
6     Option  95    2.9  Put

Index Type  Price
7     TBill 99
```

Create a new field `Maturity` for the cash instrument.

```
MDate = datenum('7/1/99');
ISet = instsetfield(ISet, 'Type', 'TBill', 'FieldName',...
'Maturity','FieldClass', 'date', 'Data', MDate);
instdisp(ISet)

Index Type  Price  Maturity
7     TBill 99     01-Jul-1999
```

Create a new field `Contracts` for all instruments.

```
ISet = instsetfield(ISet, 'FieldName', 'Contracts', 'Data', 0);
instdisp(ISet)

Index Type    Strike Price Opt  Contracts
1     Option  95    12.2  Call 0
2     Option 100     9.2  Call 0
3     Option 105     6.8  Call 0

Index Type    Delivery       F     Contracts
4     Futures 01-Jul-1999   104.4 0

Index Type    Strike Price Opt  Contracts
5     Option 105     7.4  Put  0
6     Option  95     2.9  Put  0

Index Type  Price  Maturity     Contracts
7     TBill 99     01-Jul-1999  0
```

Set the Contracts fields for some instruments.

```
ISet = instsetfield(ISet,'Index',[3; 5; 4; 7],...
'FieldName','Contracts',  'Data', [1000; -1000; -1000; 6]);

instdisp(ISet)

Index Type    Strike Price Opt  Contracts
1     Option  95    12.2  Call    0
2     Option 100     9.2  Call    0
3     Option 105     6.8  Call  1000

Index Type    Delivery       F     Contracts
4     Futures 01-Jul-1999    104.4 -1000

Index Type    Strike Price Opt  Contracts
5     Option 105     7.4  Put  -1000
6     Option  95     2.9  Put     0

Index Type  Price  Maturity    Contracts
7     TBill 99     01-Jul-1999 6
```

See Also   instaddfield, instdisp, instget, instgetcell

Purpose	Construct swap instrument

Syntax

```
InstSet = instswap(InstSet, LegRate, Settle, Maturity, LegReset,
    Basis, Principal, LegType)
[FieldList, ClassList, TypeString] = instswap
```

Arguments

InstSet	Instrument variable. This argument is specified only when adding a swap to an existing instrument set. See instget for more information on the InstSet variable.
LegRate	Number of instruments (NINST)-by-2 matrix, with each row defined as: [CouponRate Spread] or [Spread CouponRate] CouponRate is the decimal annual rate. Spread is the number of basis points over the reference rate. The first column represents the receiving leg, while the second column represents the paying leg.
Settle	Settlement date. NINST-by-1 vector of serial date numbers or date strings. Settle must be earlier than or equal to Maturity.
Maturity	Maturity date. NINST-by-1 vector of dates representing the maturity date for each swap.
LegReset	(Optional) NINST-by-2 matrix representing the reset frequency per year for each swap. Default = [1 1].
Basis	(Optional) NINST-by-1 vector representing the basis used when annualizing the input forward rate tree. Default = 0 (actual/actual).
Principal	(Optional) NINST-by-1 vector of the notional principal amounts. Default = 100.
LegType	(Optional) NINST-by-2 matrix. Each row represents an instrument. Each column indicates if the corresponding leg is fixed (1) or floating (0). This matrix defines the interpretation of the values entered in LegRate. Default is [1,0] for each instrument.

instswap

Data arguments are number of instruments (NINST)-by-1 vectors, scalar, or empty. Fill unspecified entries in vectors with NaN. Only one data argument is required to create the instrument; the others may be omitted or passed as empty matrices [].

Description   InstSet = instswap(InstSet, LegRate, Settle, Maturity, LegReset, Basis, Principal, LegType) creates a new instrument set containing swap instruments or adds swap instruments to an existing instrument set.

[FieldList, ClassList, TypeString] = instswap displays the classes.

FieldList is a number of fields (NFIELDS)-by-1 cell array of strings listing the name of each data field for this instrument type.

ClassList is an NFIELDS-by-1 cell array of strings listing the data class of each field. The class determines how arguments are parsed. Valid strings are 'dble', 'date', and 'char'.

TypeString is a string specifying the type of instrument added. For a swap instrument, TypeString = 'Swap'.

See Also   hjmprice, instaddfield, instbond, instcap, instdisp, instfloor, intenvprice

Purpose   List types

Syntax   TypeList = insttypes(InstSet)

Arguments   InstSet   Variable containing a collection of instruments. Instruments are classified by type; each type can have different data fields. The stored data field is a row vector or string for each instrument.

Description   TypeList = insttypes(InstSet) retrieves a list of types stored in an instrument variable.

TypeList is a number of types (NTYPES)-by-1 cell array of strings listing the Type of instruments contained in the variable.

Examples   Retrieve the instrument set variable ExampleInst from the data file. InstSetExamples.mat. ExampleInst contains three types of instruments: Option, Futures, and TBill.

```
load InstSetExamples;
instdisp(ExampleInst)

Index Type   Strike Price Opt  Contracts
1     Option 95   12.2 Call     0
2     Option 100   9.2 Call     0
3     Option 105   6.8 Call  1000

Index Type    Delivery        F      Contracts
4     Futures 01-Jul-1999   104.4 -1000

Index Type   Strike Price Opt  Contracts
5     Option 105    7.4 Put  -1000
6     Option 95     2.9 Put      0

Index Type Price Maturity      Contracts
7     TBill 99    01-Jul-1999  6
```

insttypes

List all of the types included in `ExampleInst`.

```
TypeList = insttypes(ExampleInst)
TypeList =
          'Futures'
          'Option'
          'TBill'
```

See Also   instdisp, instfields, instlength

Purpose	Obtain properties of an interest term structure

Syntax   ParameterValue = intenvget(RateSpec, 'ParameterName')

Arguments

RateSpec	A structure encapsulating the properties of an interest rate structure. See intenvset for information on creating RateSpec.
ParameterName	String indicating the parameter name to be accessed. The value of the named parameter is extracted from the structure RateSpec. It is sufficient to type only the leading characters that uniquely identify the parameter. Case is ignored for parameter names.

Description   ParameterValue = intenvget(RateSpec,'ParameterName') obtains the value of the named parameter ParameterName extracted from RateSpec.

Examples   Use intenvset to set the interest rate structure.

```
RateSpec = intenvset('Rates', 0.05, 'StartDates',...
'20-Jan-2000', 'EndDates', '20-Jan-2001')
```

Now use intenvget to extract the values from RateSpec.

```
[R, RateSpec] = intenvget(RateSpec, 'Rates')

R =

    0.0500
```

```
RateSpec =

        FinObj: 'RateSpec'
        Compounding: 2
        Disc: 0.9518
        Rates: 0.0500
        EndTimes: 2
        StartTimes: 0
        EndDates: 730871
        StartDates: 730505
        ValuationDate: 730505
        Basis: 0
        EndMonthRule: 1
```

See Also   intenvset

Purpose   Price fixed income instruments by a set of zero curves

Syntax   Price = intenvprice(RateSpec, InstSet)

Arguments   RateSpec   A structure encapsulating the properties of an interest
                                       rate structure. See intenvset for information on
                                       creating RateSpec.

                   InstSet   Variable containing a collection of instruments.
                                       Instruments are categorized by type; each type can have
                                       different data fields. The stored data field is a row vector
                                       or string for each instrument.

Description   Price = intenvprice(RateSpec, InstSet) computes arbitrage free prices
                   for instruments against a set of zero coupon bond rate curves.

                   Price is a number of instruments (NINST) by number of curves (NUMCURVES)
                   matrix of prices of each instrument. If an instrument cannot be priced, a NaN is
                   returned in that entry.

                   intenvprice handles the following instrument types: 'Bond', 'CashFlow',
                   'Fixed', 'Float', 'Swap'. See instadd for information about constructing
                   defined types.

                   See single-type pricing functions to retrieve pricing information.

                   bondbyzero   Price bonds by a set of zero curves.

                   cfbyzero   Price arbitrary cash flow instrument by a set of zero curves.

                   fixedbyzero   Fixed rate note prices by zero curves.

                   floatbyzero   Floating rate note prices by zero curves.

                   swapbyzero   Swap prices by a set of zero curves.

Examples   Load the zero curves and instruments from a data file.

                       load deriv.mat
                       instdisp(ZeroInstSet)

```
Index Type CouponRate Settle        Maturity     Period ...                        Name     Quantity
1     Bond 0.04       01-Jan-2000   01-Jan-2003  1                                 4% bond 100
2     Bond 0.04       01-Jan-2000   01-Jan-2004  2                                 4% bond  50

Index Type  CouponRate Settle       Maturity      FixedReset Basis Principal Name     Quantity
3     Fixed 0.04       01-Jan-2000   01-Jan-2003  1          NaN   NaN       4% Fixed 80

Index Type  Spread Settle       Maturity      FloatReset Basis Principal Name       Quantity
4     Float 20      01-Jan-2000  01-Jan-2003  1          NaN   NaN       20BP Float 8

Index Type LegRate    Settle       Maturity    LegReset Basis Principal LegType Name       Quantity
5     Swap [0.06 20] 01-Jan-2000 01-Jan-2003 [1  1]   NaN   NaN       NaN     6%/20BP Swap 10
```

```
Price = intenvprice(ZeroRateSpec, ZeroInstSet)

Price =

    98.7159
    97.5334
    98.7159
   100.5529
     3.6923
```

See Also   hjmprice, hjmsens, instadd, intenvsens, intenvset

Purpose	Instrument price and sensitivities by a set of zero curves
Syntax	`[Delta, Gamma, Price] = intenvsens(RateSpec, InstSet)`

Arguments

RateSpec	A structure encapsulating the properties of an interest rate structure. See `intenvset` for information on creating RateSpec.
InstSet	Variable containing a collection of instruments. Instruments are categorized by type; each type can have different data fields. The stored data field is a row vector or string for each instrument.

Description `[Delta, Gamma, Price] = intenvsens(RateSpec, InstSet)` computes dollar prices and price sensitivities for instruments using a zero coupon bond rate term structure.

Delta is a number of instruments (NINST) by number of curves (NUMCURVES) matrix of deltas, representing the rate of change of instrument prices with respect to shifts in the observed forward yield curve. Delta is computed by finite differences.

Gamma is an NINST-by-NUMCURVES matrix of gammas, representing the rate of change of instrument deltas with respect to shifts in the observed forward yield curve. Gamma is computed by finite differences.

Note Both sensitivities are returned as dollar sensitivities. To find the per-dollar sensitivities, divide by the respective instrument price.

Price is an NINST-by-NUMCURVES matrix of prices of each instrument. If an instrument cannot be priced, a NaN is returned.

intenvsens handles the following instrument types: `'Bond'`, `'CashFlow'`, `'Fixed'`, `'Float'`, `'Swap'`. See `instadd` for information about constructing defined types.

Examples Load the tree and instruments from a data file.

```
load deriv.mat
instdisp(ZeroInstSet)
```

Index	Type	CouponRate	Settle	Maturity	Period ...		Name	Quantity
1	Bond	0.04	01-Jan-2000	01-Jan-2003	1		4% bond	100
2	Bond	0.04	01-Jan-2000	01-Jan-2004	2		4% bond	50

Index	Type	CouponRate	Settle	Maturity	FixedReset	Basis	Principal	Name	Quantity
3	Fixed	0.04	01-Jan-2000	01-Jan-2003	1	NaN	NaN	4% Fixed	80

Index	Type	Spread	Settle	Maturity	FloatReset	Basis	Principal	Name	Quantity
4	Float	20	01-Jan-2000	01-Jan-2003	1	NaN	NaN	20BP Float	8

Index	Type	LegRate	Settle	Maturity	LegReset	Basis	Principal	LegType	Name	Quantity
5	Swap	[0.06 20]	01-Jan-2000	01-Jan-2003	[1 1]	NaN	NaN	NaN	6%/20BP Swap	10

```
[Delta, Gamma] = intenvsens(ZeroRateSpec, ZeroInstSet)

Delta =

 -272.6403
 -347.4386
 -272.6403
   -1.0445
 -282.0405


Gamma =

  1.0e+003 *

    1.0298
    1.6227
    1.0298
    0.0033
    1.0596
```

See Also   hjmprice, hjmsens, instadd, intenvprice, intenvset

Purpose   Set properties of interest rate environment

Syntax   [RateSpec, RateSpecOld] = intenvset(RateSpec, 'Parameter1', Value1,
                   'Parameter2', Value2, ...)
                 [RateSpec, RateSpecOld] = intenvset
                 intenvset

Arguments   RateSpec   (Optional) An existing interest rate specification
                                   structure to be changed, probably created from a
                                   previous call to intenvset.

                 Parameters may be chosen from the table below and specified in any order.

                 Compounding   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.

                 Disc   Number of points (NPOINTS) by number of curves
                                   (NCURVES) matrix of unit bond prices over investment
                                   intervals from StartDates, when the cash flow is valued,
                                   to EndDates, when the cash flow is received.

                 Rates   Number of points (NPOINTS) by number of curves
                                   (NCURVES) matrix of rates in decimal form. For
                                   example, 5% is 0.05 in Rates. Rates are the yields over
                                   investment intervals from StartDates, when the cash
                                   flow is valued, to EndDates, when the cash flow is
                                   received.

EndDates	NPOINTS-by-1 vector or scalar of serial maturity dates ending the interval to discount over.
StartDates	NPOINTS-by-1 vector or scalar of serial dates starting the interval to discount over. Default = ValuationDate.
ValuationDate	(Optional) Scalar value in serial date number form representing the observation date of the investment horizons entered in StartDates and EndDates. Default = min(StartDates).
Basis	(Optional) Day-count basis of the bond. A vector of integers. 0 = actual/actual (default), 1 = 30/360, 2 = actual/360, 3 = actual/365.
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.

It is sufficient to type only the leading characters that uniquely identify the parameter. Case is ignored for parameter names.

When creating a new RateSpec, the set of parameters passed to intenvset must include StartDates, EndDates, and either Rates or Disc.

Call intenvset with no input or output arguments to display a list of parameter names and possible values.

Description   [RateSpec, RateSpecOld] = intenvset(RateSpec, 'Parameter1', Value1, 'Parameter2', Value2, ...) creates an interest term structure (RateSpec) in which the input argument list is specified as parameter name /parameter value pairs. The parameter name portion of the pair must be recognized as a valid field of the output structure RateSpec; the parameter value portion of the pair is then assigned to its paired field.

If the optional argument RateSpec is specified, intenvset modifies an existing interest term structure RateSpec by changing the named parameters to the specified values and recalculating the parameters dependent on the new values.

[RateSpec, RateSpecOld] = intenvset creates an interest term structure RateSpec with all fields set to [].

intenvset with no input or output arguments displays a list of parameter names and possible values.

RateSpecOld is a structure encapsulating the properties of an interest rate structure prior to the changes introduced by the call to intenvset.

Examples   Use intenvset to create a RateSpec.

```
RateSpec = intenvset('Rates', 0.05, 'StartDates',...
'20-Jan-2000', 'EndDates', '20-Jan-2001')

RateSpec =

            FinObj: 'RateSpec'
       Compounding: 2
              Disc: 0.9518
             Rates: 0.0500
          EndTimes: 2
        StartTimes: 0
          EndDates: 730871
        StartDates: 730505
     ValuationDate: 730505
             Basis: 0
       EndMonthRule: 1
```

Now change the Compounding parameter to 1 (annual).

```
RateSpec = intenvset(RateSpec, 'Compounding', 1)
```

```
RateSpec =

            FinObj: 'RateSpec'
       Compounding: 1
              Disc: 0.9518
             Rates: 0.0506
          EndTimes: 1
        StartTimes: 0
          EndDates: 730871
        StartDates: 730505
     ValuationDate: 730505
             Basis: 0
       EndMonthRule: 1
```

Calling intenvset with no input or output arguments displays a list of parameter names and possible values.

```
intenvset

         Compounding: [ 1 | {2} | 3 | 4 | 6 | 12 | 365 | -1 ]
                Disc: [ scalar | vector (NPOINTS x 1) ]
               Rates: [ scalar | vector (NPOINTS x 1) ]
            EndDates: [ scalar | vector (NPOINTS x 1) ]
          StartDates: [ scalar | vector (NPOINTS x 1) ]
       ValuationDate: [ scalar ]
               Basis: [ {0} | 1 | 2 | 3 ]
         EndMonthRule: [ 0 | {1} ]
```

See Also   intenvget

Purpose   True if financial structure type or financial object class

Syntax   IsFinObj = isafin(Obj, ClassName)

Arguments   Obj   Name of a financial structure.

                   ClassName   String containing name of financial structure class.

Description   IsFinObj = isafin(Obj, ClassName) is True (1) if the input argument is a
                   financial structure type or financial object class.

Examples   load deriv.mat
                       IsFinObj = isafin(HJMTree, 'HJMFwdTree')

                       IsFinObj =

                               1

See Also   classfin

mkbush

Purpose   Create bushy tree

Syntax   [Tree, NumStates] = mkbush(NumLevels, NumChild, NumPos, Trim,
                 NodeVal)

Arguments

NumLevels	Number of time levels of the tree.
NumChild	1 by number of levels (NUMLEVELS) vector with number of branches (children) of the nodes in each level.
NumPos	1-by-NUMLEVELS vector containing the length of the state vectors in each time level.
Trim	Scalar 0 or 1. If Trim = 1, NumPos decreases by 1 when moving from one time level to the next. Otherwise, if Trim = 0 (Default), NumPos does not decrease.
NodeVal	Initial value at each node of the tree. Default = NaN.

Description [Tree, NumStates] = mkbush(NumLevels, NumChild, NumPos, Trim,
NodeVal) creates a bushy tree Tree with initial values NodeVal at each node.
NumStates is a 1-by-NUMLEVELS vector containing the number of state vectors in
each level.

Examples   Create a tree with four time levels, two branches per node, and a vector of three
elements in each node with each element initialized to NaN.

```
Tree = mkbush(4, 2, 3);
treeviewer(Tree)
```
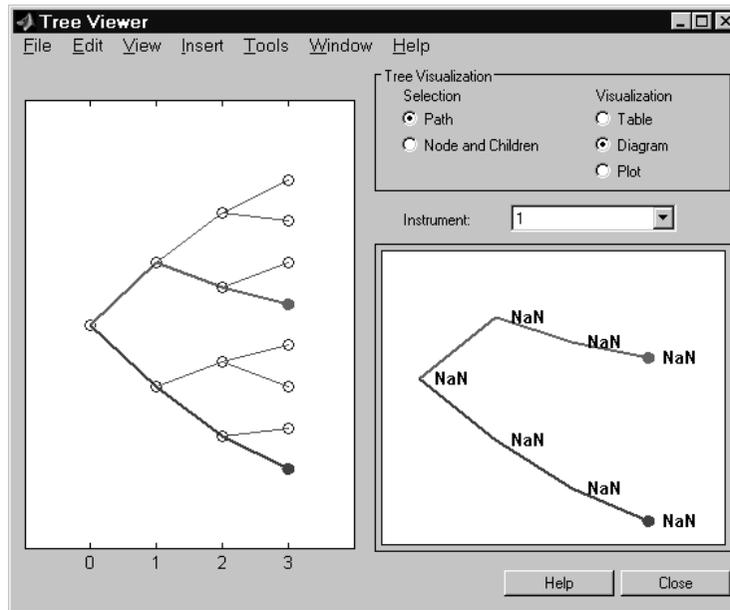

See Also   bushpath, bushshape

mktree

Purpose   Create recombining tree

Syntax   Tree = mktree(NumLevels, NumPos, NodeVal, IsPriceTree)

Arguments

NumLevels	Number of time levels of the tree.
NumPos	1-by-NUMLEVELS vector containing the length of the state vectors in each time level.
NodeVal	Initial value at each node of the tree. Default = NaN.
IsPriceTree	Boolean determining if a final horizontal branch is added to the tree. Default = 0

Description   Tree = mktree(NumLevels, NumPos, NodeVal, IsPriceTree) creates a
recombining tree Tree with initial values NodeVal at each node.

Examples   Create a recombining tree with four time levels, a vector of two elements in
each node, each element initialized to NaN.

    Tree = mktree(4, 2)

See Also   treepath, treeshape

Purpose   Create money market tree from BDT

Syntax   MMktTree = mmktbybdt(BDTTree)

Arguments   BDTTree   Interest rate tree structure created by bdttree.

Description MMktTree = mmktbybdt(BDTTree) creates a money market tree from an
                 interest rate tree structure created by bdttree.

Examples
```
load deriv.mat;
MMktTree = mmktbybdt(BDTTree);
treeviewer(MMktTree)
```
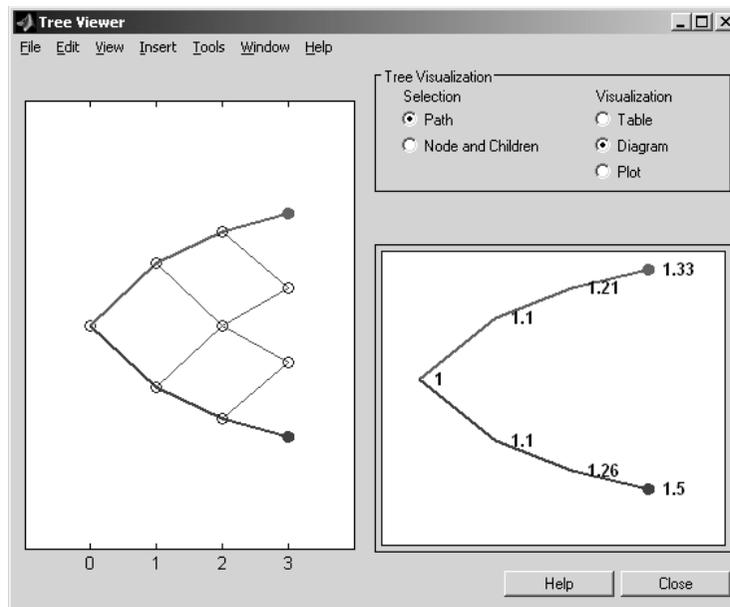

See Also   bdttree

mmktbyhjm

Purpose   Create money market tree from HJM

Syntax   MMktTree = mmktbyhjm(HJMTree)

Arguments   HJMTree   Forward rate tree structure created by hjmtree.

Description   MMktTree = mmktbyhjm(HJMTree) creates a money market tree from a forward
                    rate tree structure created by hjmtree.

Examples
```
load deriv.mat;
MMktTree = mmktbyhjm(HJMTree);
treeviewer(MMktTree)
```


See Also   hjmtree

Purpose	Price bond option by BDT interest rate tree	

Syntax	[Price, PriceTree] = optbndbybdt(BDTree, OptSpec, Strike,
	  ExerciseDates, AmericanOpt, CouponRate, Settle, Maturity,
	  Period, Basis, EndMonthRule, IssueDate, FirstCouponDate,
	  LastCouponDate, StartDate, Face, Options)	

Arguments	BDTTree	Forward rate tree structure created by hjmtree.
	OptSpec	Number of instruments (NINST)-by-1 cell array of string values 'Call' or 'Put'.
	Strike	For a European or Bermuda option: Number of instruments (NINST) by number of strikes (NSTRIKES) matrix of strike price values. Each row is the schedule for one option. If an option has fewer than NSTRIKES exercise opportunities, the end of the row is padded with NaNs. For an American option: NINST-by-1 vector of strike price values for each option.
	ExerciseDates	For a European or Bermuda option: NINST-by-NSTRIKES matrix of exercise dates. Each row is the schedule for one option. A European option has only one exercise date, the option expiry date. For an American option: NINST-by-2 vector of exercise date boundaries. For each instrument, the option can be exercised on any coupon date between or including the pair of dates on that row. If only one non-NaN date is listed, or if ExerciseDates is NINST-by-1, the option can be exercised between the underlying bond Settle and the single listed exercise date.
	AmericanOpt	NINST-by-1 vector of flags: 0 (European/Bermuda) or 1 (American).
	CouponRate	Decimal annual rate.

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 1, 2, 3, 4, 6, and 12. Default = 2.
Basis	(Optional) Day-count basis of the bond. A vector of integers. 0 = actual/actual (default), 1 = 30/360, 2 = actual/360, 3 = actual/365.
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	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	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 is followed only by the bond's maturity cash flow date.
StartDate	Ignored.

Face	Face value. Default is 100.
Options	(Optional) Derivatives pricing options structure created with `derivset`.

The `Settle` date for every bond is set to the `ValuationDate` of the BDT tree. The bond argument `Settle` is ignored.

Description   `[Price, PriceTree] = optbndbybdt(BDTree, OptSpec, Strike, ExerciseDates, AmericanOpt, CouponRate, Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate, StartDate, Face, Options)` computes the price of a bond option from a BDT interest rate tree.

`Price` is an NINST-by-1 matrix of expected prices at time 0.

`PriceTree` is a tree structure with a vector of instrument prices at each node.

Examples   Example 1.

Using the BDT interest rate tree in the `deriv.mat` file, price a European call option on a 10% bond with a strike of 95. The exercise date for the option is Jan. 01, 2002. The settle date for the bond is Jan. 01, 2000, and the maturity date is Jan. 01, 2003.

Load the file `deriv.mat`, which provides `BDTTree`. `BDTTree` contains the time and forward rate information needed to price the bond.

```
load deriv;
```

Use `optbondbybdt` to compute the price of the option.

```
Price = optbndbybdt(BDTTree,'Call','95','01-Jan-2002',...
'0','0.10','01-Jan-2000','01-Jan-2003','1')

Price =

    1.7657
```

Example 2.

Now use optbndbybdt to compute the price of a put option on the same bond.

```
Price = optbndbybdt(BDTTree,'Put','95','01-Jan-2002',...
'0','0.10','01-Jan-2000','01-Jan-2003','1')

Price =

    0.5740
```

See Also   bdtprice, bdttree, instoptbnd

Purpose   Price bond option by HJM interest rate tree

Syntax   [Price, PriceTree] = optbndbyhjm(HJMTree, OptSpec, Strike,
                       ExerciseDates, AmericanOpt, CouponRate, Settle, Maturity,
                       Period, Basis, EndMonthRule, IssueDate, FirstCouponDate,
                       LastCouponDate, StartDate, Face, Options)

Arguments

HJMTree	Forward rate tree structure created by hjmtree.
OptSpec	Number of instruments (NINST)-by-1 cell array of string values 'Call' or 'Put'.
Strike	For a European or Bermuda option: Number of instruments (NINST) by number of strikes (NSTRIKES) matrix of strike price values. Each row is the schedule for one option. If an option has fewer than NSTRIKES exercise opportunities, the end of the row is padded with NaNs.

For an American option: NINST-by-1 vector of strike price values for each option. |
| ExerciseDates | For a European or Bermuda option: NINST-by-NSTRIKES matrix of exercise dates. Each row is the schedule for one option. A European option has only one exercise date, the option expiry date.

For an American option: NINST-by-2 vector of exercise date boundaries. For each instrument, the option can be exercised on any coupon date between or including the pair of dates on that row. If only one non-NaN date is listed, or if ExerciseDates is NINST-by-1, the option can be exercised between the underlying bond Settle and the single listed exercise date. |
| AmericanOpt | NINST-by-1 vector of flags: 0 (European/Bermuda) or 1 (American). |
| CouponRate | Decimal annual rate. |

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 1, 2, 3, 4, 6, and 12. Default = 2.
Basis	(Optional) Day-count basis of the bond. A vector of integers. 0 = actual/actual (default), 1 = 30/360, 2 = actual/360, 3 = actual/365.
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 is followed only by the bond's maturity cash flow date.
StartDate	Ignored.

Face   (Optional) Face value. Default = 100.

Options   (Optional) Derivatives pricing options structure created
                    with `derivset`.

The `Settle` date for every bond is set to the `ValuationDate` of the HJM tree.
The bond argument `Settle` is ignored.

Description   `[Price, PriceTree] = optbndbyhjm(HJMTree, OptSpec, Strike, ExerciseDates, AmericanOpt, CouponRate, Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate, StartDate, Face, Options)` computes the price of a bond option from an HJM forward rate tree.

`Price` is an NINST-by-1 matrix of expected prices at time 0.

`PriceTree` is a tree structure with a vector of instrument prices at each node.

Examples   Using the HJM forward rate tree in the `deriv.mat` file, price a European call option on a 4% bond with a strike of 96. The exercise date for the option is Jan. 01, 2003. The settle date for the bond is Jan. 01, 2000, and the maturity date is Jan. 01, 2004.

Load the file `deriv.mat`, which provides `HJMTree`. `HJMTree` contains the time and forward rate information needed to price the bond.

```
load deriv;
```

Use `optbondbyhjm` to compute the price of the option.

```
Price = optbndbyhjm(HJMTree,'Call','96','01-Jan-2003',...
'0','0.04','01-Jan-2000','01-Jan-2004')
Warning: Not all cash flows are aligned with the tree. Result will
be approximated.

Price =

    2.2410
```

See Also   `hjmprice, hjmtree, instoptbnd`

rate2disc

Purpose   Discounting factors from interest rates

Syntax   Usage 1: Interval points are input as times in periodic units.

```
Disc = rate2disc(Compounding, Rates, EndTimes, StartTimes)
```

Usage 2: ValuationDate is passed and interval points are input as dates.

```
[Disc, EndTimes, StartTimes] = rate2disc(Compounding, Rates,
   EndDates, StartDates, ValuationDate)
```

Arguments

Compounding	Scalar value representing the rate at which the input zero rates were compounded when annualized. This argument determines the formula for the discount factors: Compounding = 1, 2, 3, 4, 6, 12 Disc = $(1 + Z/F)^{-T}$, where F is the compounding frequency, Z is the zero rate, and T is the time in periodic units, e.g. T = F is one year. Compounding = 365 Disc = $(1 + Z/F)^{-T}$, where F is the number of days in the basis year and T is a number of days elapsed computed by basis. Compounding = -1 Disc = $\exp(-T*Z)$, where T is time in years.
Rates	Number of points (NPOINTS) by number of curves (NCURVES) matrix of rates in decimal form. For example, 5% is 0.05 in Rates. Rates are the yields over investment intervals from StartTimes, when the cash flow is valued, to EndTimes, when the cash flow is received.
EndTimes	NPOINTS-by-1 vector or scalar of times in periodic units ending the interval to discount over.
StartTimes	(Optional) NPOINTS-by-1 vector or scalar of times in periodic units starting the interval to discount over. Default = 0.

EndDates	NPOINTS-by-1 vector or scalar of serial maturity dates ending the interval to discount over.
StartDates	(Optional) NPOINTS-by-1 vector or scalar of serial dates starting the interval to discount over. Default = ValuationDate.
ValuationDate	Scalar value in serial date number form representing the observation date of the investment horizons entered in StartDates and EndDates. Required in Usage 2. Omitted or passed as an empty matrix to invoke Usage 1.

Description   Disc = rate2disc(Compounding, Rates, EndTimes, StartTimes) and [Disc, EndTimes, StartTimes] = rate2disc(Compounding, Rates, EndDates, StartDates, ValuationDate) convert interest rates to cash flow discounting factors. rate2disc computes the discounts over a series of NPOINTS time intervals given the annualized yield over those intervals. NCURVES different rate curves can be translated at once if they have the same time structure. The time intervals can represent a zero curve or a forward curve.

Disc is an NPOINTS-by-NCURVES column vector of discount factors in decimal form representing the value at time StartTime of a unit cash flow received at time EndTime.

StartTimes is an NPOINTS-by-1 column vector of times starting the interval to discount over, measured in periodic units.

EndTimes is an NPOINTS-by-1 column vector of times ending the interval to discount over, measured in periodic units.

If Compounding = 365 (daily), StartTimes and EndTimes are measured in days. The arguments otherwise contain values, T, computed from SIA semiannual time factors, Tsemi, by the formula T = Tsemi/2*F, where F is the compounding frequency.

The investment intervals can be specified either with input times (Usage 1) or with input dates (Usage 2). Entering ValuationDate invokes the date interpretation; omitting ValuationDate invokes the default time interpretations.

rate2disc

Example 1.

Compute discounts from a zero curve at six months, 12 months, and 24 months.
The time to the cash flows is 1, 2, and 4. We are computing the present value
(at time 0) of the cash flows.

```
Compounding = 2;
Rates = [0.05; 0.06; 0.065];
EndTimes   = [1; 2; 4];
Disc = rate2disc(Compounding, Rates, EndTimes)

Disc =
    0.9756
    0.9426
    0.8799
```

Example 2.

Compute discounts from a zero curve at six months, 12 months, and 24 months.
Use dates to specify the ending time horizon.

```
Compounding = 2;
Rates = [0.05; 0.06; 0.065];
EndDates = ['10/15/97'; '04/15/98'; '04/15/99'];
ValuationDate = '4/15/97';
Disc = rate2disc(Compounding, Rates, EndDates, [], ValuationDate)

Disc =
    0.9756
    0.9426
    0.8799
```

Example 3.

Compute discounts from the one-year forward rates beginning now, in six
months, and in 12 months. Use monthly compounding. The times to the cash
flows are 12, 18, 24, and the forward times are 0, 6, 12.

```
Compounding = 12;
Rates = [0.05; 0.04; 0.06];
EndTimes = [12; 18; 24];
StartTimes = [0; 6; 12];
Disc = rate2disc(Compounding, Rates, EndTimes, StartTimes)
```

```
Disc =
    0.9513
    0.9609
    0.9419
```

See Also   disc2rate, ratetimes

ratetimes

Purpose	Change time intervals defining interest rate environment

Syntax

Usage 1: `ValuationDate` not passed; third through sixth arguments are interpreted as times.

```
[Rates, EndTimes, StartTimes] = ratetimes(Compounding, RefRates,
    RefEndTimes, RefStartTimes, EndTimes, StartTimes)
```

Usage 2: `ValuationDate` passed and interval points input as dates.

```
[Rates, EndTimes, StartTimes] = ratetimes(Compounding, RefRates,
    RefEndDates, RefStartDates, EndDates, StartDates, ValuationDate)
```

Arguments

Compounding	Scalar value representing the rate at which the input zero rates were compounded when annualized. This argument determines the formula for the discount factors: Compounding = 1, 2, 3, 4, 6, 12 Disc = $(1 + Z/F)^{-T}$, where F is the compounding frequency, Z is the zero rate, and T is the time in periodic units, e.g. T = F is one year. Compounding = 365 Disc = $(1 + Z/F)^{-T}$, where F is the number of days in the basis year and T is a number of days elapsed computed by basis. Compounding = -1 Disc = $\exp(-T*Z)$, where T is time in years.
RefRates	NREFPTS-by-NCURVES matrix of reference rates in decimal form. `RefRates` are the yields over investment intervals from `RefStartTimes`, when the cash flow is valued, to `RefEndTimes`, when the cash flow is received.
RefEndTimes	NREFPTS-by-1 vector or scalar of times in periodic units ending the intervals corresponding to `RefRates`.
RefStartTimes	(Optional) NREFPTS-by-1 vector or scalar of times in periodic units starting the intervals corresponding to `RefRates`. Default = 0.

EndTimes	NPOINTS-by-1 vector or scalar of times in periodic units ending the interval to discount over.
StartTimes	(Optional) NPOINTS-by-1 vector or scalar of times in periodic units starting the interval to discount over. Default = 0.
RefEndDates	NREFPTS-by-1 vector or scalar of serial dates ending the intervals corresponding to RefRates.
RefStartDates	(Optional) NREFPTS-by-1 vector or scalar of serial dates starting the intervals corresponding to RefRates. Default = ValuationDate.
EndDates	NPOINTS-by-1 vector or scalar of serial maturity dates ending the interval to discount over.
StartDates	(Optional) NPOINTS-by-1 vector or scalar of serial dates starting the interval to discount over. Default = ValuationDate.
ValuationDate	Scalar value in serial date number form representing the observation date of the investment horizons entered in StartDates and EndDates. Required in usage 2. Omitted or passed as an empty matrix to invoke usage 1.

Description   [Rates, EndTimes, StartTimes] = ratetimes(Compounding, RefRates, RefEndTimes, RefStartTimes, EndTimes, StartTimes) and
[Rates, EndTimes, StartTimes] = ratetimes(Compounding, RefRates, RefEndDates, RefStartDates, EndDates, StartDates, ValuationDate)
change time intervals defining an interest rate environment.

ratetimes takes an interest rate environment defined by yields over one collection of time intervals and computes the yields over another set of time intervals. The zero rate is assumed to be piecewise linear in time.

Rates is an NPOINTS-by-NCURVES matrix of rates implied by the reference interest rate structure and sampled at new intervals.

StartTimes is an NPOINTS-by-1 column vector of times starting the new intervals where rates are desired, measured in periodic units.

EndTimes is an NPOINTS-by-1 column vector of times ending the new intervals, measured in periodic units.

If Compounding = 365 (daily), StartTimes and EndTimes are measured in days. The arguments otherwise contain values, T, computed from SIA semiannual time factors, Tsemi, by the formula T = Tsemi/2 * F, where F is the compounding frequency.

The investment intervals can be specified either with input times (Usage 1) or with input dates (Usage 2). Entering the argument ValuationDate invokes the date interpretation; omitting ValuationDate invokes the default time interpretations.

Examples   Example 1.

The reference environment is a collection of zero rates at six, 12, and 24 months. Create a collection of one year forward rates beginning at zero, six, and 12 months.

```
RefRates = [0.05; 0.06; 0.065];
RefEndTimes = [1; 2; 4];
StartTimes = [0; 1; 2];
EndTimes   = [2; 3; 4];
Rates = ratetimes(2, RefRates, RefEndTimes, 0, EndTimes,...
StartTimes)

Rates =
    0.0600
    0.0688
    0.0700
```

Example 2.

Interpolate a zero yield curve to different dates. Zero curves start at the default date of ValuationDate.

```
RefRates = [0.04; 0.05; 0.052];
RefDates = [729756; 729907; 730121];
Dates    = [730241; 730486];
ValuationDate   = 729391;
Rates = ratetimes(2, RefRates, RefDates, [], Dates, [],...
ValuationDate)
```

```
Rates =
    0.0520
    0.0520
```

See Also   disc2rate, rate2disc

swapbybdt

Purpose   Price swap instrument by BDT interest rate tree

Syntax
```
[Price, PriceTree, CFTree, SwapRate] = swapbybdt(BDTTree, LegRate,
    Settle, Maturity, LegReset, Basis, Principal, LegType, Options)
```

Arguments

BDTTree	Interest rate tree structure created by bdttree.
LegRate	Number of instruments (NINST)-by-2 matrix, with each row defined as: [CouponRate Spread] or [Spread CouponRate] CouponRate is the decimal annual rate. Spread is the number of basis points over the reference rate. The first column represents the receiving leg, while the second column represents the paying leg.
Settle	Settlement date. NINST-by-1 vector of serial date numbers or date strings. Settle must be earlier than or equal to Maturity.
Maturity	Maturity date. NINST-by-1 vector of dates representing the maturity date for each swap.
LegReset	(Optional) NINST-by-2 matrix representing the reset frequency per year for each swap. Default = [1 1].
Basis	(Optional) NINST-by-1 vector representing the basis used when annualizing the input forward rate tree. Default = 0 (actual/actual).
Principal	(Optional) NINST-by-1 vector of the notional principal amounts. Default = 100.
LegType	(Optional) NINST-by-2 matrix. Each row represents an instrument. Each column indicates if the corresponding leg is fixed (1) or floating (0). This matrix defines the interpretation of the values entered in LegRate. Default is [1 0] for each instrument.
Options	(Optional) Derivatives pricing options structure created with derivset.

The Settle date for every swap is set to the ValuationDate of the BDT tree. The swap argument Settle is ignored.

This function also calculates the SwapRate (fixed rate) so that the value of the swap is initially zero. To do this enter CouponRate as NaN.

Description   [Price, PriceTree, CFTree, SwapRate] = swapbybdt(BDTTree, LegRate, Settle, Maturity, LegReset, Basis, Principal, LegType) computes the price of a swap instrument from a BDT interest rate tree.

Price is number of instruments (NINST)-by-1 expected prices of the swap at time 0.

PriceTree is the tree structure with a vector of the swap values at each node.

CFTree is the tree structure with a vector of the swap cash flows at each node.

SwapRate is a NINST-by-1 vector of rates applicable to the fixed leg such that the swaps' values are zero at time 0. This rate is used in calculating the swaps' prices when the rate specified for the fixed leg in LegRate is NaN. SwapRate is padded with NaN for those instruments in which CouponRate is not set to NaN.

Examples   Example 1.

Price an interest rate swap with a fixed receiving leg and a floating paying leg. Payments are made once a year, and the notional principal amount is $100. The values for the remaining parameters are:

- Coupon rate for fixed leg: 0.15 (15%)
- Spread for floating leg: 10 basis points
- Swap settlement date: Jan. 01, 2000
- Swap maturity date: Jan. 01, 2003

Based on the information above, set the required parameters and build the LegRate, LegType, and LegReset matrices.

```
Settle = '01-Jan-2000';
Maturity = '01-Jan-2003';
Basis = 0;
Principal = 100;
LegRate = [0.15 10]; % [CouponRate Spread]
LegType = [1 0]; % [Fixed Float]
```

```
LegReset = [1 1]; % Payments once per year
```

Price the swap using the BDTTree included in the MAT-file deriv.mat. BDTTree contains the time and forward rate information needed to price the instrument.

```
load deriv;
```

Use swapbybdt to compute the price of the swap.

```
Price  = swapbybdt(BDTTree, LegRate, Settle, Maturity,...
LegReset, Basis, Principal, LegType)

Price =

   7.3032
```

Example 2.

Using the previous data, calculate the swap rate, the coupon rate for the fixed leg such that the swap price at time = 0 is zero.

```
LegRate = [NaN 20];

[Price, PriceTree, CFTree, SwapRate] = swapbybdt(BDTTree,...
LegRate, Settle, Maturity, LegReset, Basis, Principal, LegType)

Price =

 -2.8422e-014

PriceTree =

    FinObj: 'BDTPriceTree'
      tObs: [0 1 2 3 4]
     PTree: {1x5 cell}

CFTree =

    FinObj: 'BDTCFTree'
      tObs: [0 1 2 3 4]
     CFTree: {1x5 cell}
```

```
SwapRate =

    0.1210
```

Example 3.

Calculate the cash flows from a pair of swaps and display the result.

```
Settle = '01-Jan-2000';
Maturity = '01-Jan-2003';
Basis = 0;
Principal = 100;
LegRate= [0.15 10; 0.15 0]; % [CouponRate Spread]
LegType = [1 0; 1 0];
LegReset = [1 1; 1 1];

load deriv

[Price, PriceTree, CFTree, SwapRate] = swapbybdt(BDTTree,...
LegRate, Settle, Maturity, LegReset, Basis, Principal, LegType);
```

Continuing on, provide names for the swaps. Then use `treeviewer` to observe the cash flow data graphically.

```
Names ={'Swap1', 'Swap2'};
treeviewer(CFTree, Names)
```
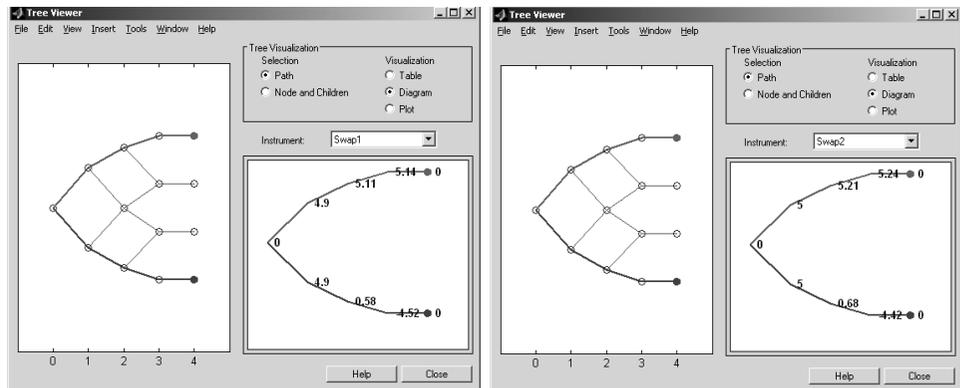
You can use `treeviewer` to display cash flow data at all observation times and along all branches of the tree.

See Also   `bdttree`, `capbybdt`, `cfbybdt`, `floorbybdt`

Purpose	Price swap instrument by HJM interest rate tree	

Syntax   [Price, PriceTree, CFTree, SwapRate] = swapbyhjm(HJMTree, LegRate,
        Settle, Maturity, LegReset, Basis, Principal, LegType, Options)

Arguments

HJMTree	Forward rate tree structure created by hjmtree.
LegRate	Number of instruments (NINST)-by-2 matrix, with each row defined as: [CouponRate Spread] or [Spread CouponRate] CouponRate is the decimal annual rate. Spread is the number of basis points over the reference rate. The first column represents the receiving leg, while the second column represents the paying leg.
Settle	Settlement date. NINST-by-1 vector of serial date numbers or date strings. Settle must be earlier than or equal to Maturity.
Maturity	Maturity date. NINST-by-1 vector of dates representing the maturity date for each swap.
LegReset	(Optional) NINST-by-2 matrix representing the reset frequency per year for each swap. Default = [1 1].
Basis	(Optional) NINST-by-1 vector representing the basis used when annualizing the input forward rate tree. Default = 0 (actual/actual).
Principal	(Optional) NINST-by-1 vector of the notional principal amounts. Default = 100.
LegType	(Optional) NINST-by-2 matrix. Each row represents an instrument. Each column indicates if the corresponding leg is fixed (1) or floating (0). This matrix defines the interpretation of the values entered in LegRate. Default is [1 0] for each instrument.
Options	(Optional) Derivatives pricing options structure created with derivset.

swapbyhjm

The Settle date for every swap is set to the ValuationDate of the HJM tree. The swap argument Settle is ignored.

This function also calculates the SwapRate (fixed rate) so that the value of the swap is initially zero. To do this enter CouponRate as NaN.

Description [Price, PriceTree, CFTree, SwapRate] = swapbyhjm(HJMTree, LegRate, Settle, Maturity, LegReset, Basis, Principal, LegType) computes the price of a swap instrument from an HJM interest rate tree.

Price is number of instruments (NINST)-by-1 expected prices of the swap at time 0.

PriceTree is the tree structure with a vector of the swap values at each node.

CFTree is the tree structure with a vector of the swap cash flows at each node.

SwapRate is a NINST-by-1 vector of rates applicable to the fixed leg such that the swaps' values are zero at time 0. This rate is used in calculating the swaps' prices when the rate specified for the fixed leg in LegRate is NaN. SwapRate is padded with NaN for those instruments in which CouponRate is not set to NaN.

Examples Example 1.

Price an interest rate swap with a fixed receiving leg and a floating paying leg. Payments are made once a year, and the notional principal amount is $100. The values for the remaining parameters are:

- Coupon rate for fixed leg: 0.06 (6%)
- Spread for floating leg: 20 basis points
- Swap settlement date: Jan. 01, 2000
- Swap maturity date: Jan. 01, 2003

Based on the information above, set the required parameters and build the LegRate, LegType, and LegReset matrices.

```
Settle = '01-Jan-2000';
Maturity = '01-Jan-2003';
Basis = 0;
Principal = 100;
LegRate = [0.06 20]; % [CouponRate Spread]
LegType = [1 0]; % [Fixed Float]
```

```
LegReset = [1 1]; % Payments once per year
```

Price the swap using the `HJMTree` included in the MAT-file `deriv.mat`. `HJMTree` contains the time and forward rate information needed to price the instrument.

```
load deriv;
```

Use `swapbyhjm` to compute the price of the swap.

```
[Price, PriceTree, CFTree] = swapbyhjm(HJMTree, LegRate,...
Settle, Maturity, LegReset, Basis, Principal, LegType)

Price =

    3.6923

PriceTree =

     FinObj: 'HJMPriceTree'
       tObs: [0 1 2 3 4]
      PBush: {1x5 cell}

CFTree =

     FinObj: 'HJMCFTree'
       tObs: [0 1 2 3 4]
      CFBush: {[0] [1x1x2 double] [1x2x2 double] ... [1x8 double]}
```

Using the function `treeviewer`, you can examine `CFTree` graphically and see the cash flows from the swap along both the up and the down branches. A positive cash flow indicates an inflow (income - payments > 0), while a negative cash flow indicates an outflow (income - payments < 0).

```
treeviewer(CFTree)
```
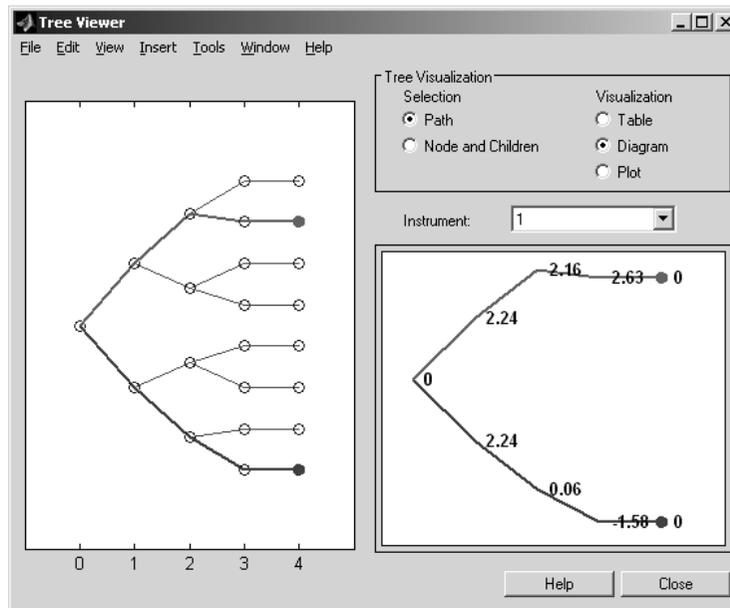
Note `treeviewer` price tree diagrams follow the convention that increasing prices appear on the upper branch of a tree and, consequently, decreasing prices appear on the lower branch. Conversely, for interest rate displays, *decreasing* interest rates appear on the upper branch (prices are rising) and *increasing* interest rates on the lower branch (prices are falling).

In this example you have sold a swap (receive fixed and pay floating). At time $t = 3$, if interest rates go down, your cash flow is positive ($2.63), meaning that you will receive this amount. But if interest rates go up, your cash flow is negative(-$1.58), meaning that you owe this amount.

Example 2.

Using the previous data, calculate the swap rate, the coupon rate for the fixed leg such that the swap price at time = 0 is zero.

```
LegRate = [NaN 20];

[Price, PriceTree, CFTree, SwapRate] = swapbyhjm(HJMTree,...
LegRate, Settle, Maturity, LegReset, Basis, Principal, LegType)

Price =

   0

PriceTree =

FinObj: 'HJMPriceTree'
  tObs: [0 1 2 3 4]
 PBush:{[0] [1x1x2 double] [1x2x2 double] ... [1x8 double]}

CFTree =

FinObj: 'HJMCFTree'
  tObs: [0 1 2 3 4]
CFBush:{[0] [1x1x2 double] [1x2x2 double] ... [1x8 double]}

SwapRate =

   0.0466
```

See Also   capbyhjm, cfbyhjm, floorbyhjm, hjmtree

swapbyzero

Purpose	Price swap instrument by a set of zero curves	
Syntax	[Price, SwapRate] = swapbyzero(RateSpec, LegRate, Settle, Maturity, LegReset, Basis, Principal, LegType)	
Arguments	RateSpec	A structure encapsulating the properties of an interest rate structure. See intenvset for information on creating RateSpec.
	LegRate	Number of instruments (NINST)-by-2 matrix, with each row defined as: [CouponRate Spread] or [Spread CouponRate] CouponRate is the decimal annual rate. Spread is the number of basis points over the reference rate. The first column represents the receiving leg, while the second column represents the paying leg.
	Settle	Settlement date. NINST-by-1 vector of serial date numbers or date strings representing the settlement date for each swap. Settle must be earlier than or equal to Maturity.
	Maturity	Maturity date. NINST-by-1 vector of dates representing the maturity date for each swap.
	LegReset	(Optional) NINST-by-2 matrix representing the reset frequency per year for each swap. Default = [1 1].
	Basis	(Optional) NINST-by-1 vector representing the basis used when annualizing the input forward rate tree. Default = 0 (actual/actual).
	Principal	(Optional) NINST-by-1 vector of the notional principal amounts. Default = 100.
	LegType	(Optional) NINST-by-2 matrix. Each row represents an instrument. Each column indicates if the corresponding leg is fixed (1) or floating (0). This matrix defines the interpretation of the values entered in LegRate. Default is [1 0] for each instrument.

Description   [Price, SwapRate] = swapbyzero(RateSpec, LegRate, Settle, Maturity, LegReset, Basis, Principal, LegType) prices a swap instrument by a set of zero coupon bond rates.

Price is a NINST by number of curves (NUMCURVES) matrix of swap prices. Each column arises from one of the zero curves.

SwapRate is an NINST-by-NUMCURVES matrix of rates applicable to the fixed leg such that the swap's values are zero at time 0. This rate is used in calculating the swaps' prices when the rate specified for the fixed leg in LegRate is NaN. SwapRate is padded with NaN for those instruments in which CouponRate is not set to NaN.

Examples   Example 1.

Price an interest rate swap with a fixed receiving leg and a floating paying leg. Payments are made once a year, and the notional principal amount is $100. The values for the remaining parameters are:

- Coupon rate for fixed leg: 0.06 (6%)
- Spread for floating leg: 20 basis points
- Swap settlement date: Jan. 01, 2000
- Swap maturity date: Jan. 01, 2003

Based on the information above, set the required parameters and build the LegRate, LegType, and LegReset matrices.

```
Settle = '01-Jan-2000';
Maturity = '01-Jan-2003';
Basis = 0;
Principal = 100;
LegRate = [0.06 20]; % [CouponRate Spread]
LegType = [1 0]; % [Fixed Float]
LegReset = [1 1]; % Payments once per year
```

Load the file deriv.mat, which provides ZeroRateSpec, the interest rate term structure needed to price the bond.

```
load deriv
```

Use swapbyzero to compute the price of the swap.

```
Price = swapbyzero(ZeroRateSpec, LegRate, Settle, Maturity,...
LegReset, Basis, Principal, LegType)

Price =
   3.6923
```

Example 2.

Using the previous data, calculate the swap rate, the coupon rate for the fixed leg such that the swap price at time = 0 is zero.

```
LegRate = [NaN 20];

[Price, SwapRate] = swapbyzero(ZeroRateSpec, LegRate, Settle,...
Maturity, LegReset, Basis, Principal, LegType)

Price =
   0

SwapRate =
   0.0466
```

See Also   bondbyzero, cfbyzero, fixedbyzero, floatbyzero

Purpose	Extract entries from node of recombining tree

Syntax   `Values = treepath(Tree, BranchList)`

Arguments

`Tree`	Recombining tree.
`BranchList`	Number of paths (`NUMPATHS`) by path length (`PATHLENGTH`) matrix containing the sequence of branchings.

Description `Values = treepath(Tree, BranchList)` extracts entries of a node of a recombining tree. The node path is described by the sequence of branchings taken, starting at the root. The top branch is number one, the second-to-top is two, and so on. Set the branch sequence to zero to obtain the entries at the root node.

`Values` is a number of values (`NUMVALS`)-by-`NUMPATHS` matrix containing the retrieved entries of a recombining tree.

Examples Create a BDT tree by loading the example file.

```
load deriv.mat;
```

Then

```
FwdRates = treepath(BDTTree.FwdTree, [1 2 1])
```

returns the rates at the tree nodes located by taking the up branch, then the down branch, and finally the up branch again.

```
FwdRates =

    1.1000
    1.0979
    1.1377
    1.1183
```

You can visualize this with the `treeviewer` function.

```
treeviewer(BDTTree)
```


See Also   mktree, treeshape

Purpose	Retrieve shape of a recombining tree

Syntax   [NumLevels, NumPos, IsPriceTree] = treeshape(Tree)

Arguments   Tree   Recombining tree.

Description   [NumLevels, NumPos, IsPriceTree] = treeshape(Tree) returns
information on a recombining tree's shape.

NumLevels is the number of time levels of the tree.

NumPos is a 1-by-NUMLEVELS vector containing the length of the state vectors in
each level.

IsPriceTree is a Boolean determining if a final horizontal branch is present in
the tree.

Examples   Create a BDT tree by loading the example file.

```
load deriv.mat;
```

With treeviewer you can see the general shape of the BDT interest rate tree.

treeshape

```
treeviewer(BDTTree)
```
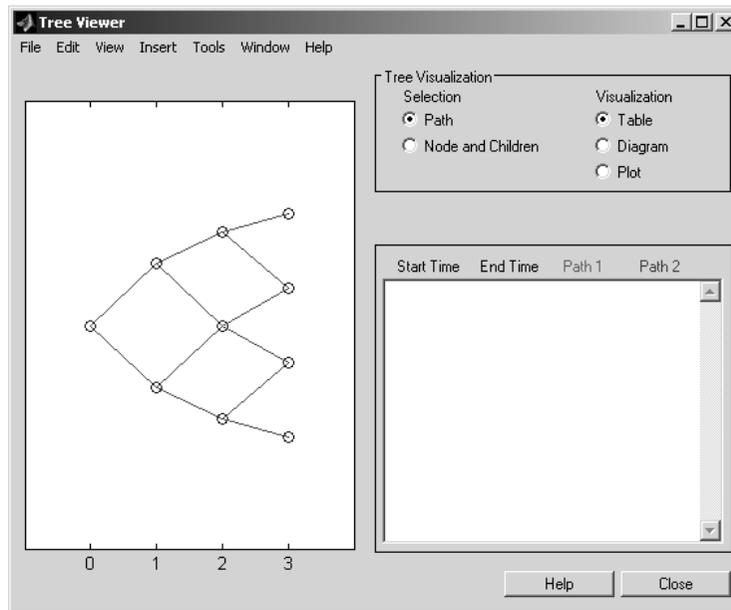

With this tree

```
[NumLevels, NumPos, IsPriceTree] = treeshape(BDTTree.FwdTree)
```

returns

```
NumLevels =
     4

NumPos    =
     1    1    1    1

IsPriceTree =
     0
```

See Also   mktree, treepath

Purpose   Display tree information

Syntax   treeviewer(Tree)
                   treeviewer(PriceTree, InstSet)
                   treeviewer(CFTree, InstSet)

Arguments

Tree	Tree can be a Black-Derman-Toy (BDTTree) or Heath-Jarrow-Morton (HJMTree) interest rate tree or a money market tree (MMktTree). See bdttree for information on creating BDTTree. See hjmtree for information on creating HJMTree. The functions mmktbybdt and mmktbyhjm create money market trees.
PriceTree	PriceTree is a Black-Derman-Toy (BDTPriceTree) or Heath-Jarrow-Morton (HJMPriceTree) tree of instrument prices.
CFTree	CFTree is a BDT (BDTCFTree) or HJM (HJMCFTree) tree of swap cash flows. You create cash flow trees when executing the functions swapbybdt and swapbyhjm.
InstSet	(Optional) Variable containing a collection of instruments whose prices or cash-flows are contained in a tree. The collection can be created with the function instadd or as a cell array containing the names of the instruments. To display the names of the instruments, the field Name should exist in InstSet. If InstSet is not passed, treeviewer uses default instruments names (numbers) when displaying prices or cash flows.

Description   treeviewer(Tree) displays an interest rate or money market tree.

treeviewer(PriceTree, InstSet) displays a tree of instrument prices. If you provide the name of an instrument set (InstSet) and you have named the instruments using the field Name, the treeviewer display identifies the instrument being displayed with its name. (See Example 3 below for a description.) If you do not provide the optional InstSet argument, the instruments are identified by their sequence number in the instrument set. (See Example 6 below for a description.)

treeviewer(CFTree, InstSet) displays a cash flow tree that has been created with swapbybdt or swapbyhjm. If you provide the name of an instrument set (InstSet) containing cash flow names, the treeviewer display identifies the instrument being displayed with its name. (See Example 3 below for a description.) If the optional InstSet argument is not present, the instruments are identified by their sequence number in the instrument set. See Example 6 below for a description.)

treeviewer price tree diagrams follow the convention that increasing prices appear on the upper branch of a tree and, consequently, decreasing prices appear on the lower branch. Conversely, for interest rate displays, *decreasing* interest rates appear on the upper branch (prices are rising) and *increasing* interest rates on the lower branch (prices are falling).

treeviewer provides an interactive display of prices or interest rates. The display is activated by clicking on the nodes along the price or interest rate path shown in the left panel when the function is called. For HJM trees you select the end points of the path, and treeviewer displays all data from beginning to end. With BDT trees you must click on *each* node in succession from the beginning (t = 1) to the last node (t = n). Do not include the *root node*, the node at t = 0. If you do not click on the nodes in the proper order, you are reminded with the message:

```
Parent of selected node must be selected.
```

Examples
Example 1.
Display an HJM interest rate tree.

```
load deriv.mat
treeviewer(HJMTree)
```

The treeviewer function displays the structure of an HJM tree in the left panel. The tree visualization in the right panel is blank.
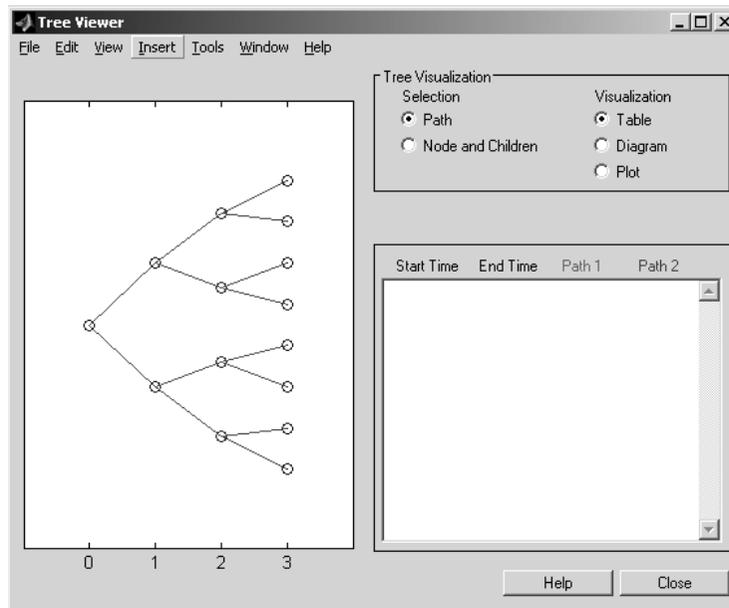
To visualize the actual interest rate tree, go to the **Tree Visualization** panel and click on **Path** (the default) and **Diagram.** Now, select the first path by clicking on the last node (t = 3) of the upper branch.
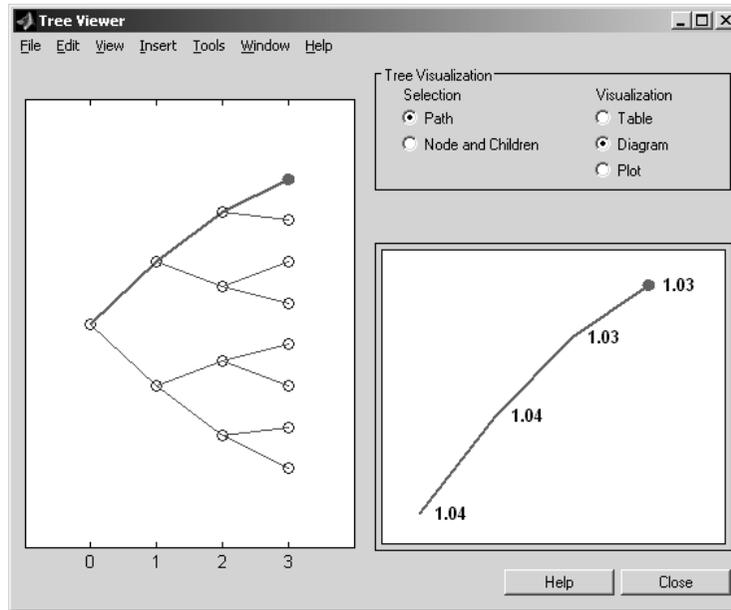
Note that the entire upper path is highlighted in red.

To complete the process, select a second path by clicking on the last node
(t = 3) of another branch. The second path is highlighted in purple. The final
display looks like
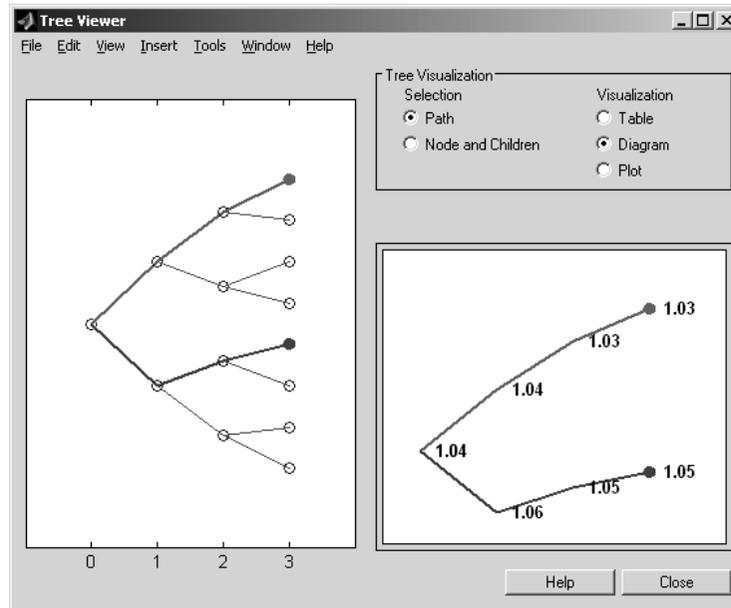
Alternative Forms of Display

The **Tree Visualization** panel allows you to select alternative ways to display tree data. For example, if you select **Path** and **Table** as your visualization choices, the final display above instead appears in tabular form.
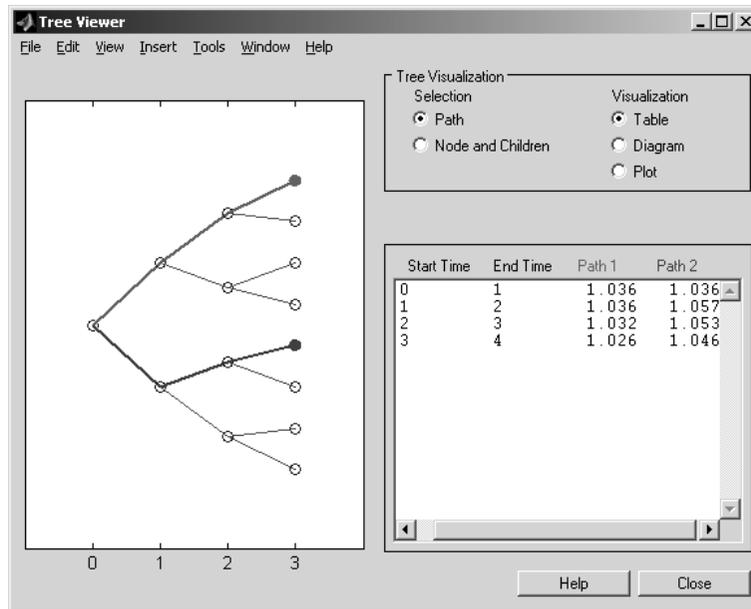
To see a plot of interest rates along the chosen branches, choose **Path** and **Plot** in the **Tree Visualization** panel.
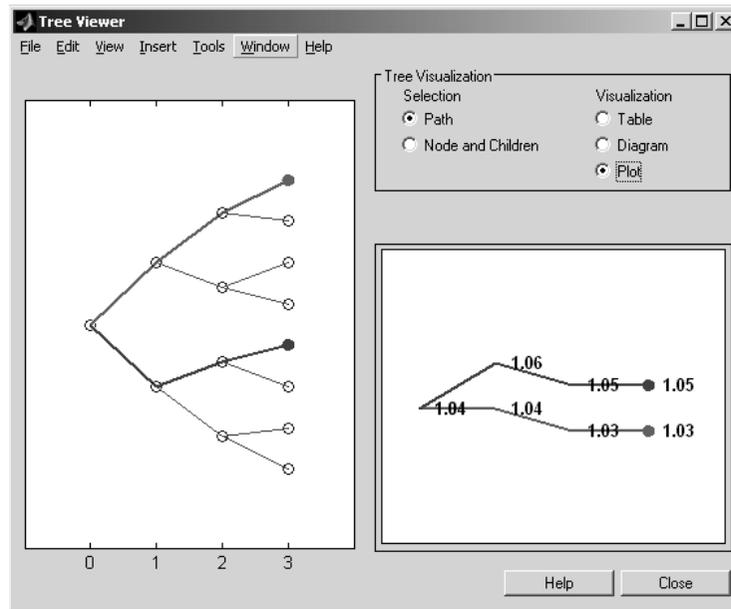
Note that with **Plot** selected rising interest rates are shown on the upper branch and declining interest rates on the lower.

Finally, if you choose **Node and Children** under **Tree Visualization**, you restrict the data displayed to just the selected parent node and its children.
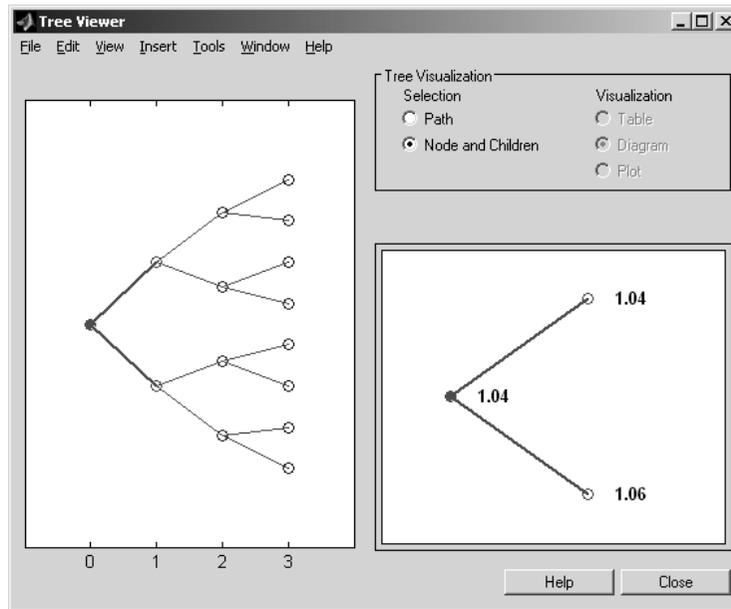
With **Node and Children** selected, the choices under **Visualization** are unavailable.

Example 2.
Display a BDT interest rate tree.

```
load deriv.mat
treeviewer(BDTTree)
```

The treeviewer function displays the structure of a BDT tree in the left panel.
The tree visualization in the right panel is blank.
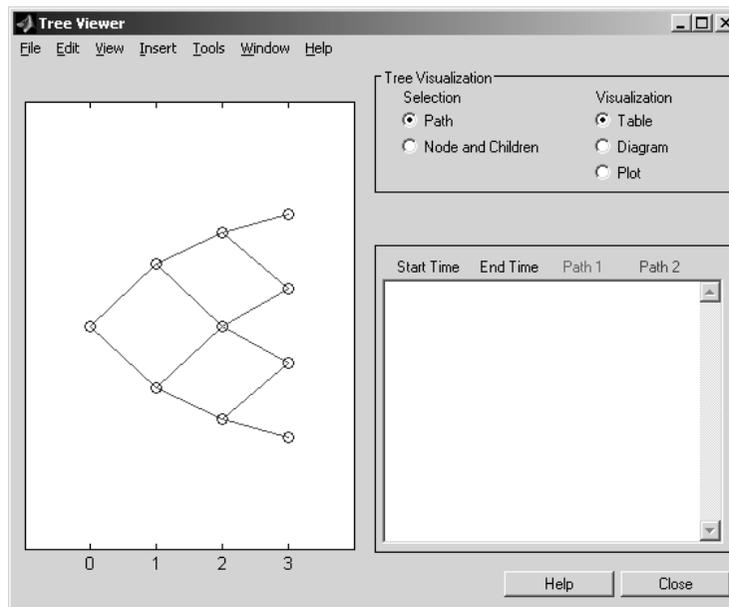
To visualize the actual interest rate tree, go to the **Tree Visualization** panel
and click on **Path** (the default) and **Diagram.** Now, select the first path by
clicking on the first node of the up branch (t = 1). Continue by clicking on the
down branch at the next node (t = 2). The two figures below show the
treeviewer path diagrams for these selections.
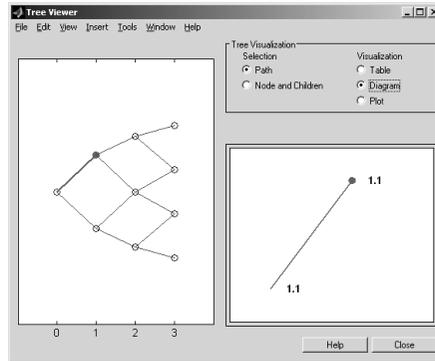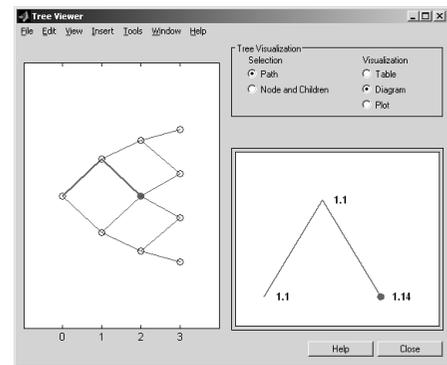
t = 1   t = 2

Continue clicking on all nodes in succession until you reach the end of the branch. Note that the entire path you have selected is highlighted in red.

Select a second path by clicking on the first node of the lower branch (t = 1). Continue clicking on lower nodes as you did on the first branch. Note that the second branch is highlighted in purple. The final display looks like
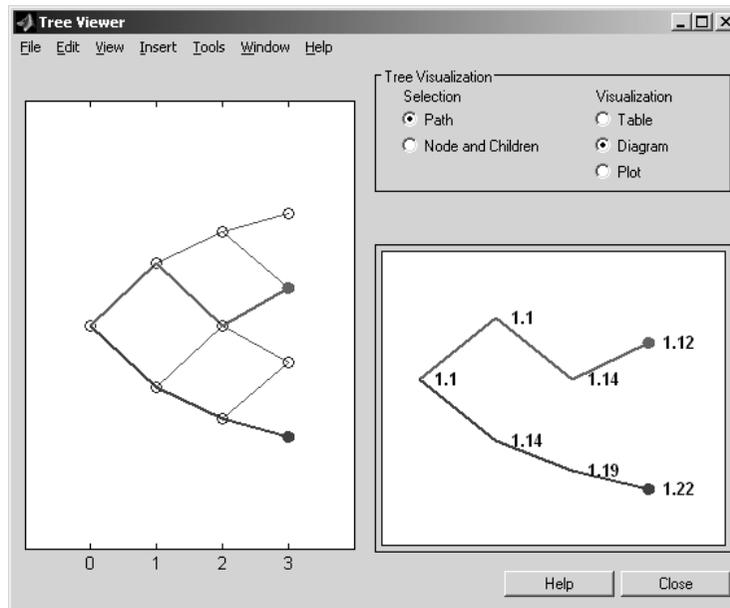
Example 3.
Display an HJM price tree for named instruments.

```
load deriv.mat
[Price, PriceTree] = hjmprice(HJMTree, HJMInstSet);
treeviewer(PriceTree, HJMInstSet)
```
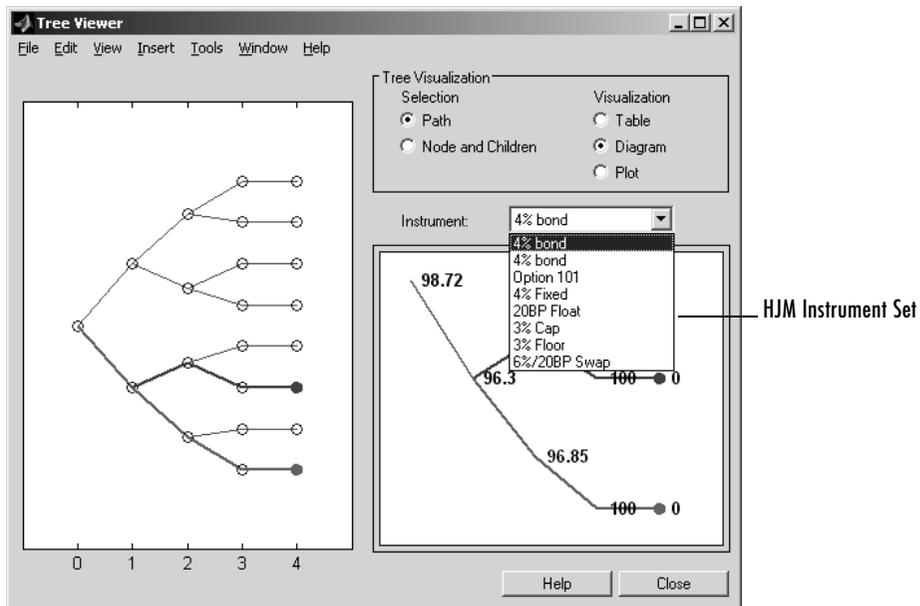

HJM Instrument Set

Example 4.
Display a BDT price tree for named instruments.

```
load deriv.mat
[Price, PriceTree] = bdtprice(BDTTree, BDTInstSet);
treeviewer(PriceTree, BDTInstSet)
```
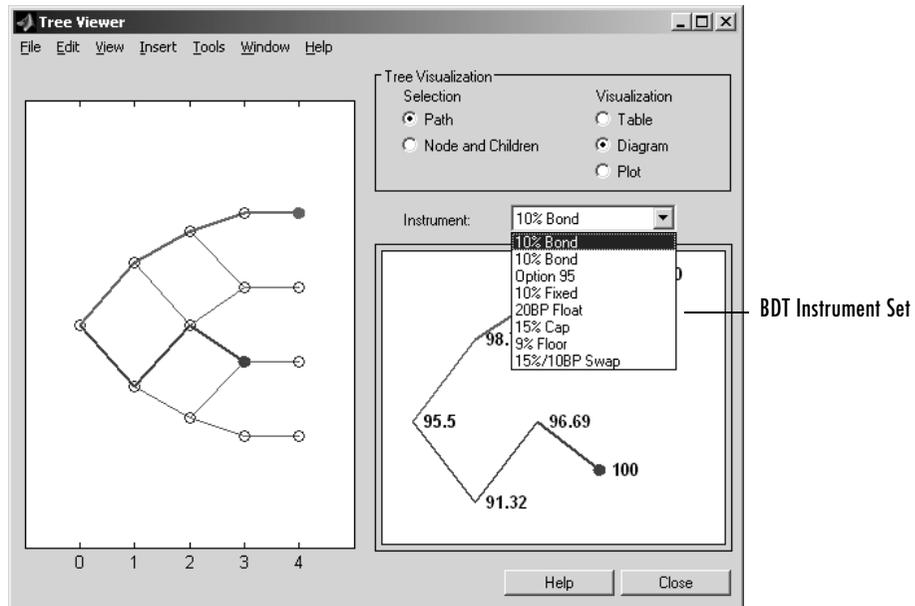

BDT Instrument Set

Example 5.

Display an HJM price tree with renamed instruments.

```
load deriv.mat
[Price, PriceTree] = hjmprice(HJMTree, HJMInstSet);
Names = {'Bond1', 'Bond2', 'Option', 'Fixed','Float', 'Cap',...
'Floor', 'Swap'};
treeviewer(PriceTree, Names)
```
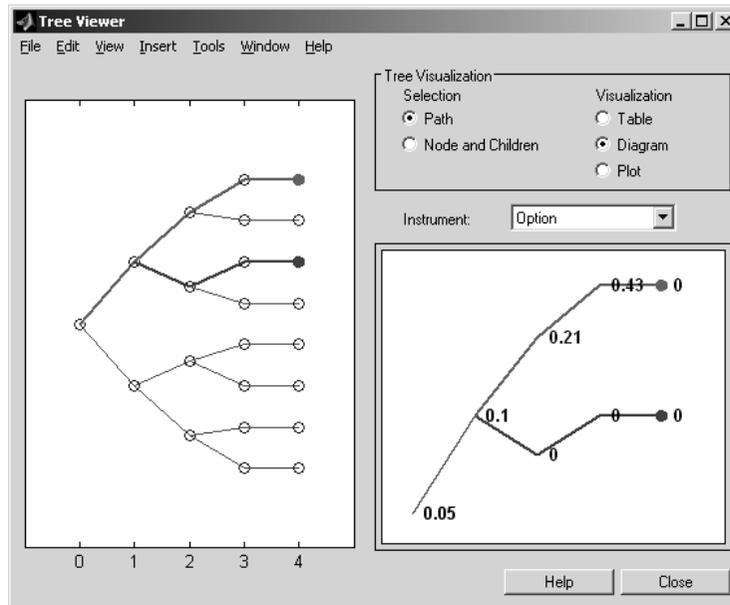
Example 6.
Display an HJM price tree using default instrument names (numbers).

```
load deriv.mat
[Price, PriceTree] = hjmprice(HJMTree, HJMInstSet);
treeviewer(PriceTree)
```
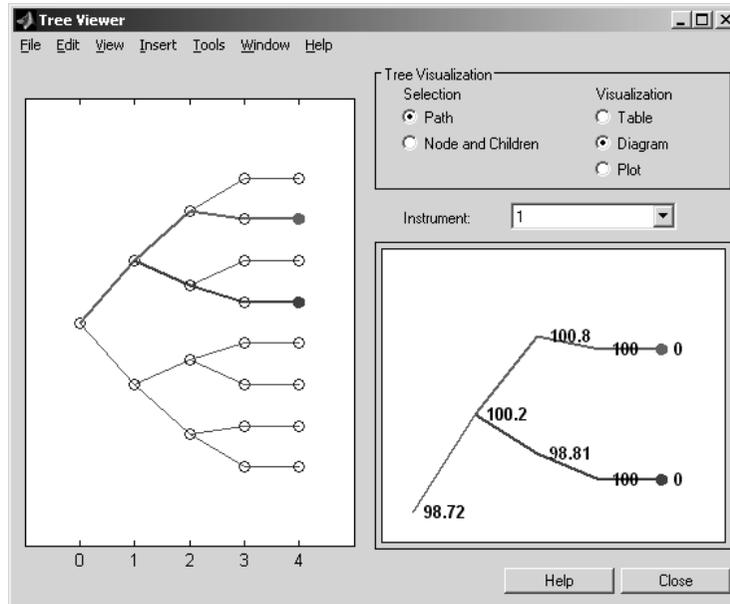

See Also   bdttree, hjmtree, instadd, mmktbybdt, mmktbyhjm, swapbybdt, swapbyhjm

Glossary

call option; i.e., the holder has sold a call option to the issuer. See **puttable bond**.

cap - Interest-rate option that guarantees that the rate on a floating-rate loan will not exceed a certain level.

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.

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.

deterministic model - An interest rate model in which the values of the rates in the next time step are determined solely by the values of the rates in the current time step.

discount factor - Coefficient used to compute the present value of future cash flows.

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.

fixed rate note - A long-term debt security with preset interest rate and maturity, by which the interest must be paid. The principal may or may not be paid at maturity.

floating rate note - A security similar to a bond, but in which the note's interest rate is reset periodically, relative to a reference index rate, to reflect fluctuations in market interest rates.

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.

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.

Heath-Jarrow-Morton (HJM) model - A model of the interest rate term structure that works with a type of interest rate tree called a **bushy tree**.

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

instrument set - A collection of financial assets. A portfolio.

inverse discount - A factor by which the present value of an asset is multiplied to find its future value. The reciprocal of the discount factor.

least squares method - A mathematical method of determining the best fit of a curve to a series of observations by choosing the curve that minimizes the sum of the squares of all deviations from the curve.

long rate - The yield on a zero-coupon Treasury bond.

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.

per-dollar sensitivity - The dollar sensitivity divided by the corresponding instrument price.

portfolio - A collection of financial assets. Also called an instrument set.

price tree structure - A MATLAB structure that holds all pricing information.

price vector - A vector of instrument prices.

pricing options structure - A MATLAB structure that defines how the price tree is used to find the price of instruments in the portfolio, and how much additional information is displayed in the command window when the pricing function is called.

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**.

rate specification - A MATLAB structure that holds all information needed to identify completely the evolution of interest rates.

recombining tree - A tree of prices or interest rates whose branches recombine over time. Opposite of a **bushy tree**.

self-financing hedge - A trading strategy whereby the value of a portfolio after rebalancing is equal to its value at any previous time.

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.

short rate - The annualized one-period interest rate.

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.

stochastic model - Involving or containing a random variable or variables; involving chance or probability.

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.

time specification - A MATLAB structure that represents the mapping between times and dates for interest rate quoting.

under-determined system - A set of simultaneous equations in which the number of independent variables exceeds the number of equations in the set, leading to an infinite number of solutions.

vanilla swap - A **swap** agreement to exchange a fixed rate for a floating rate.

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 specification - A MATLAB structure that specifies the forward rate volatility process.

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.

U

V