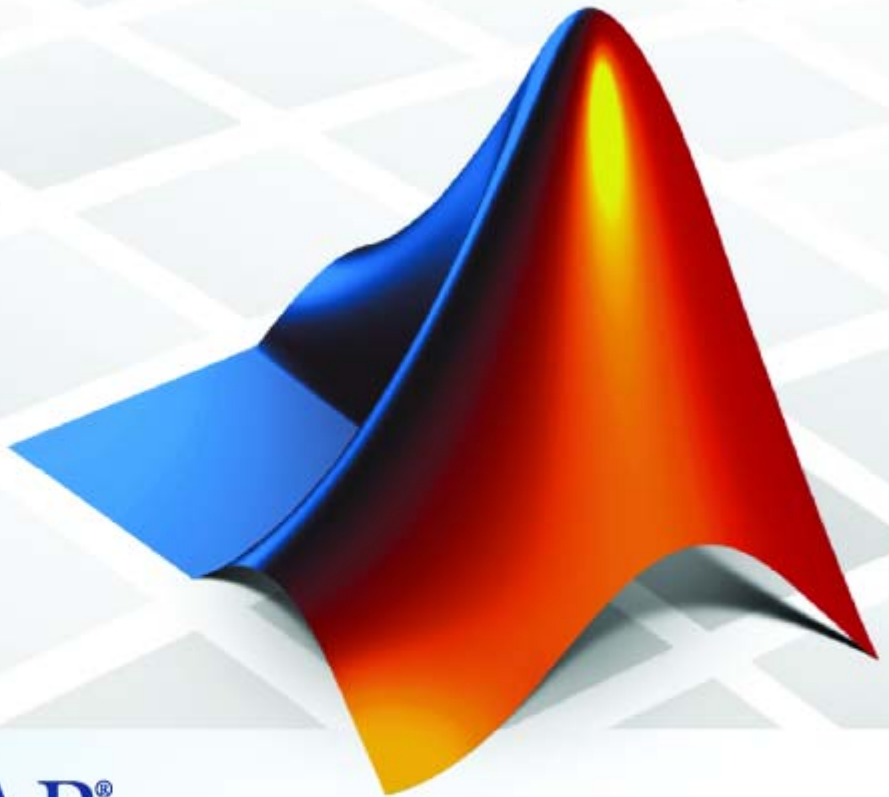


Financial Derivatives Toolbox 5

User's Guide



MATLAB[®]

How to Contact The MathWorks



www.mathworks.com
comp.soft-sys.matlab
www.mathworks.com/contact_TS.html

Web
Newsgroup
Technical Support



suggest@mathworks.com
bugs@mathworks.com
doc@mathworks.com
service@mathworks.com
info@mathworks.com

Product enhancement suggestions
Bug reports
Documentation error reports
Order status, license renewals, passcodes
Sales, pricing, and general information



508-647-7000 (Phone)



508-647-7001 (Fax)



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

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

Financial Derivatives Toolbox User's Guide

© COPYRIGHT 2000–2007 The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB, Simulink, Stateflow, Handle Graphics, Real-Time Workshop, and xPC TargetBox are registered trademarks, and SimBiology, SimEvents, and SimHydraulics are trademarks of The MathWorks, Inc.

Other product or brand names are trademarks or registered trademarks of their respective holders.

Patents

The MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

June 2000	First printing	New for Version 1.0 (Release 12)
September 2001	Second printing	Revised for Version 2.0 (Release 12.1)
April 2004	Third printing	Revised for Version 3.0 (Release 14)
September 2005	Fourth printing	Revised for Version 4.0 (Release 14SP3)
March 2006	Online only	Revised for Version 4.0.1 (Release 2006a)
September 2006	Online only	Revised for Version 4.1 (Release 2006b)
March 2007	Fifth printing	Revised for Version 5.0 (Release 2007a)

Getting Started

1

What Is Financial Derivatives Toolbox?	1-2
Interest-Rate-Based Derivatives	1-2
Equity Derivatives	1-2
Portfolio Creation	1-4
Interest-Rate-Based Derivatives	1-4
Equity Derivatives	1-5
Adding Instruments to an Existing Portfolio	1-6
Portfolio Management	1-8
Instrument Constructors	1-8
Creating New Instruments or Properties	1-9
Searching or Subsetting a Portfolio	1-11

Interest-Rate Based Financial Derivatives

2

Introduction	2-2
Financial Instruments	2-2
Interest-Rate Modeling	2-3
Rate and Price Trees	2-5
Viewing Rate or Price Movement with This Toolbox	2-6
Hedging	2-7
Understanding the Interest-Rate Term Structure	2-9
Interest Rates vs. Discount Factors	2-9
Interest-Rate Term Conversions	2-14
Functions That Model the Interest-Rate Term Structure ..	2-18
Computing Prices and Sensitivities Using the Interest-Rate Term Structure	2-23

Computing Instrument Prices	2-24
Computing Instrument Sensitivities	2-26
Understanding Interest-Rate Models	2-28
Building a Tree of Forward Rates	2-29
Specifying the Volatility Model (VolSpec)	2-31
Specifying the Interest-Rate Term Structure (RateSpec) ..	2-34
Specifying the Time Structure (TimeSpec)	2-35
Examples of Tree Creation	2-36
Examining Trees	2-37
Computing Prices and Sensitivities Using Interest-Rate Models	2-49
Computing Instrument Prices	2-49
Computing Instrument Sensitivities	2-58
Graphical Representation of Trees	2-61
Observing Interest Rates	2-61
Observing Instrument Prices	2-65

Equity Derivatives

3

Understanding Equity Trees	3-2
Building Equity Binary Trees	3-3
Building Implied Trinomial Trees	3-8
Examining Equity Trees	3-16
Differences Between CRR and EQP Tree Structures	3-20
Understanding Equity Exotic Options	3-22
Asian Option	3-22
Barrier Option	3-23
Compound Option	3-24
Lookback Option	3-24
Bermuda Put and Call Schedule	3-25
Computing Prices and Sensitivities for Equity Derivatives	3-26

Computing Instrument Prices	3-26
Computing Prices Using CRR	3-28
Computing Prices Using EQP	3-30
Computing Prices Using ITT	3-32
Examining Output from the Pricing Functions	3-34
Computing Instrument Sensitivities	3-38
Graphical Representation of CRR, EQP, and ITT Trees ...	3-42

Hedging Portfolios

4

Hedging	4-2
Hedging Functions	4-2
Hedging with hedgeopt	4-4
Self-Financing Hedges with hedgeslf	4-11
Specifying Constraints with ConSet	4-15
Setting Constraints	4-15
Portfolio Rebalancing	4-17
Hedging with Constrained Portfolios	4-20
Example: Fully Hedged Portfolio	4-20
Example: Minimize Portfolio Sensitivities	4-23
Example: Under-Determined System	4-24
Example: Portfolio Constraints with hedgeslf	4-25

Functions — By Category

5

Portfolio Hedge Allocation	5-2
Price and Sensitivity from Interest-Rate Term Structure	5-2

Price and Sensitivity from Heath-Jarrow-Morton Trees	5-3
Price and Sensitivity from Black-Derman-Toy Trees ..	5-3
Price and Sensitivity from Black-Karasinski Trees ...	5-4
Price and Sensitivity from Cox-Ross-Rubinstein Trees	5-4
Price and Sensitivity from Equal Probabilities Binomial Trees	5-5
Price and Sensitivity from Hull-White Trees	5-5
Price and Sensitivity from Implied Trinomial Tree ...	5-6
Heath-Jarrow-Morton Utilities	5-6
Black-Derman-Toy Utilities	5-7
Black-Karasinski Utilities	5-8
Cox-Ross-Rubinstein Utilities	5-8
Equal Probabilities Tree Utilities	5-9
Implied Trinomial Tree Utilities	5-9
Hull-White Utilities	5-10
Tree Manipulation	5-10
Derivatives Pricing Options	5-11
Instrument Portfolio Handling	5-11

Financial Object Structures	5-12
Interest Term Structure	5-13
Date	5-13
Graphical Display	5-13

Functions — Alphabetical List

6

Derivatives Pricing Options

A

Pricing Options Structure	A-2
Default Structure	A-2
Customizing the Structure	A-5

Bibliography

B

Black-Derman-Toy (BDT) Modeling	B-2
Heath-Jarrow-Morton (HJM) Modeling	B-3
Hull-White (HW) and Black-Karasinski (BK) Modeling	B-4
Cox-Ross-Rubinstein (CRR) Modeling	B-5

Implied Trinomial Tree (ITT) Modeling	B-6
Equal Probabilities Tree (EQP) Modeling	B-7
Financial Derivatives	B-8

Examples

C	
Instrument Portfolio Examples	C-2
Interest Rate Environment Examples	C-2
HJM Examples	C-2
Volatility Modeling	C-2
BDT Examples	C-2
Rate Specification Creation	C-3
Time Specification	C-3
Sensitivity	C-3
Treeviewer Examples	C-3
Creating Equity Derivatives	C-3
Pricing Equity Derivatives	C-4
Hedging Examples	C-4
Hedging with Constrained Portfolios	C-4

Glossary

Index

Getting Started

What Is Financial Derivatives
Toolbox? (p. 1-2)

Various types of interest-rate-based
and equity-based derivatives
handled by the toolbox

Portfolio Creation (p. 1-4)

Use of the `instadd` function to create
a new instrument portfolio or to
add new instruments to an existing
portfolio

Portfolio Management (p. 1-8)

Functions that construct new types
of financial instruments and modify
existing financial instruments and
portfolios

What Is Financial Derivatives Toolbox?

Financial Derivatives Toolbox provides components for analyzing individual derivative instruments and portfolios containing several types of interest-rate-based and equity-based financial instruments.

- “Interest-Rate-Based Derivatives” on page 1-2
- “Equity Derivatives” on page 1-2

Interest-Rate-Based Derivatives

The toolbox provides functionality that supports the creation and management of these interest-rate-based instruments:

- Bonds
- Bond options (put and call)
- Fixed-rate notes
- Floating-rate notes
- Caps
- Floors
- Swaps

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

The toolbox provides pricing and sensitivity routines for these instruments. (See “Computing Prices and Sensitivities Using the Interest-Rate Term Structure” on page 2-23 or “Computing Prices and Sensitivities Using Interest-Rate Models” on page 2-49 for information.)

Equity Derivatives

The toolbox also provides functions for the creation and management of various equity derivatives, including

- Asian options

- Barrier options
- Compound options
- Lookback options
- Vanilla stock options (put and call options)

The toolbox also provides pricing and sensitivity routines for these instruments. (See “Computing Prices and Sensitivities for Equity Derivatives” on page 3-26.)

Portfolio Creation

- “Interest-Rate-Based Derivatives” on page 1-4
- “Equity Derivatives” on page 1-5
- “Adding Instruments to an Existing Portfolio” on page 1-6

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. For interest-rate-based derivatives, the types are: `Bond`, `OptBond`, `CashFlow`, `Fixed`, `Float`, `Cap`, `Floor`, and `Swap`. For equity derivatives, the types are `Asian`, `Barrier`, `Compound`, `Lookback`, and `OptStock`.

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)
```

Interest-Rate-Based Derivatives

In addition to the bond instrument already described, the toolbox can create portfolios containing the following set of interest-rate-based derivatives:

- Bond option

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

- Arbitrary cash flow instrument

```
InstSet = instadd('CashFlow', CFflowAmounts, CFflowDates, 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)
```

Equity Derivatives

The toolbox can create portfolios containing the following set of equity derivatives:

- Asian instrument

```
InstSet = instadd('Asian', OptSpec, Strike, Settle,
ExerciseDates, AmericanOpt, AvgType, AvgPrice, AvgDate)
```

- Barrier instrument

```
InstSet = instadd('Barrier', OptSpec, Strike, Settle,
ExerciseDates, AmericanOpt, BarrierType, Barrier, Rebate)
```

- Compound instrument

```
InstSet = instadd('Compound', UOptSpec, UStrike, USettle,
UExerciseDates, UAmericanOpt, COptSpec, CStrike, CSettle,
CExerciseDates, CAmericanOpt)
```

- Lookback instrument

```
InstSet = instadd('Lookback', OptSpec, Strike, Settle,  
ExerciseDates, AmericanOpt)
```

- Stock option instrument

```
InstSet = instadd('OptStock', OptSpec, Strike, Settle, Maturity,  
AmericanOpt)
```

Adding Instruments to an Existing Portfolio

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	1	0	100
2	Cap	0.07	08-Feb-2000	15-Jan-2003	1	0	100

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	1	0	100
2	Cap	0.07	08-Feb-2000	15-Jan-2003	1	0	100

Index	Type	CouponRate	Settle	Maturity	Period	Basis	...Face
3	Bond	0.04	08-Feb-2000	15-Jan-2003	2	0	100

Portfolio Management

- “Instrument Constructors” on page 1-8
- “Creating New Instruments or Properties” on page 1-9
- “Searching or Subsetting a Portfolio” on page 1-11

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
Asian option	instasian
Barrier option	instbarrier
Bond	instbond
Bond option	instoptbnd
Arbitrary cash flow	instcf
Compound option	instcompound
Fixed-rate note	instfixed
Floating-rate note	instfloat
Cap	instcap
Floor	instfloor
Lookback option	instlookback
Stock option	instoptstock
Swap	instswap

Each instrument has parameters (fields) that describe the instrument. The toolbox functions let you

- 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`, and so on. Additionally, each instrument has a field that identifies the investment type (bond, cap, floor, and so on).

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 `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 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};
```

Finally, create the portfolio with a single instrument.

```
Port = instaddfield('Type', Type, 'FieldName', FieldName,...  
    '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	Principial
2	Cap	0.06	08-Feb-2000	15-Jan-2003	1	0	100
3	Cap	0.07	08-Feb-2000	15-Jan-2003	1	0	100

Searching or Subsetting a Portfolio

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
      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 = ...
instdisp(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(HJMIInstSet,...  
  '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(HJMIInstSet,'FieldName',...  
  'Quantity','Data',{'50';'-50'})
```

```
Pos50 =
```

```
2  
3
```

instselect

The syntax for `instselect` is 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 `HJMIInstSet` with a maturity date of January 01, 2003.

```
Mat2003 = ...  
instfind(HJMIInstSet,'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	[0.06 20]	01-Jan-2000	01-Jan-2003	[1 1]	NaN	NaN	[NaN]	6%/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.

Interest-Rate Based Financial Derivatives

Introduction (p. 2-2)

The interest-rate models used to model fixed-income derivatives. Difference between bushy and recombining trees. Binomial and trinomial trees.

Understanding the Interest-Rate Term Structure (p. 2-9)

Evolution of interest rates through time.

Computing Prices and Sensitivities Using the Interest-Rate Term Structure (p. 2-23)

Computation of prices and sensitivities using interest-rate curves.

Understanding Interest-Rate Models (p. 2-28)

Creation of HJM and BDT interest-rate models.

Computing Prices and Sensitivities Using Interest-Rate Models (p. 2-49)

Use of the models in pricing and sensitivity computations.

Graphical Representation of Trees (p. 2-61)

Use of the treeviewer function to display tree data graphically.

Introduction

Financial Derivatives Toolbox extends the capabilities of Financial Toolbox in the areas of fixed-income derivatives and of securities contingent on 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.

- “Financial Instruments” on page 2-2
- “Interest-Rate Modeling” on page 2-3
- “Rate and Price Trees” on page 2-5
- “Viewing Rate or Price Movement with This Toolbox” on page 2-6
- “Hedging” on page 2-7

Financial Instruments

The toolbox provides a set of functions that perform computations on portfolios containing up to seven types of interest-rate based financial instruments.

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

Bond Option. Puts and calls on portfolios of bonds. The toolbox supports three types of put and call options on bonds:

- American option: An option that can be exercised any time until its expiration date.
- European option: An option that can be exercised only on its expiration date.
- Bermuda option: A Bermuda option is somewhat like a hybrid of American and European options. It can be exercised on predetermined dates only, usually once a month.

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 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 that includes a guarantee that sets the maximum interest rate to be paid by the holder, based on an otherwise floating interest rate.

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

Swap. A contract between two parties obligating the parties to exchange future cash flows. This version of 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 on zero coupon bonds or on any of the various interest rate models that the toolbox supports. (See “Interest-Rate Modeling” on page 2-3.)

Interest-Rate Modeling

Financial Derivatives Toolbox computes prices and sensitivities of interest-rate contingent claims based on several methods of modeling changes in interest rates over time:

- The interest-rate term structure

This model uses sets of zero coupon bonds to predict changes in interest rates.

- Heath-Jarrow-Morton (HJM) model

The HJM 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 on a statistical process.

- Black-Derman-Toy (BDT) model

In the BDT model, all security prices and rates depend on 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. The resulting tree can then be used to determine the value of interest-rate sensitive securities from this tree.

- Hull-White (HW) model

The Hull-White model incorporates the initial term structure of interest rates and the volatility term structure to build a trinomial recombining tree of short rates. The resulting tree is used to value interest-rate dependent securities. The implementation of the HW model in Financial Derivatives Toolbox is limited to one factor.

- Black-Karasinski (BK) model

The BK model is a single-factor, log-normal version of the HW model.

For detailed information about interest-rate models, see

- “Computing Prices and Sensitivities Using the Interest-Rate Term Structure” on page 2-23 for a discussion of price and sensitivity based on portfolios of zero coupon bonds.
- “Computing Prices and Sensitivities Using Interest-Rate Models” on page 2-49 for a discussion of price and sensitivity based on the HJM and BDT interest-rate models.

Note Historically, the initial version of Financial Derivatives toolbox provided only the HJM interest-rate model. A later version added the BDT model. The current version adds both the HW and BK models. This chapter provides extensive examples of using the HJM and BDT models to compute prices and sensitivities of interest-rate based financial derivatives.

The HW and BK tree structures are similar to the BDT tree structure. To avoid needless repetition throughout this chapter, documentation is provided only where significant deviations from the BDT structure exist. Specifically, “HW and BK Tree Structures” on page 2-45 explains the few noteworthy differences among the various formats.

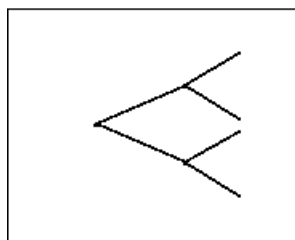
If you need more detailed information about functions that use the HW and BK tree structures, see Chapter 5, “Functions — By Category” which provides extensive reference information for all functions that compose this toolbox.

Rate and Price Trees

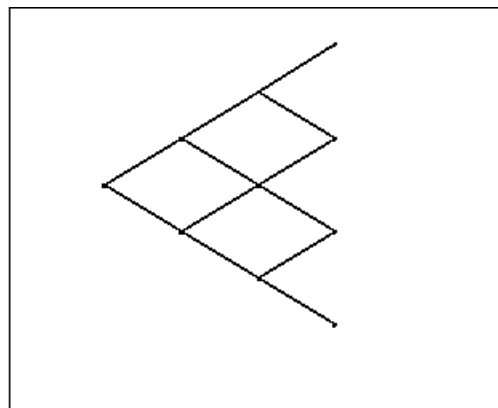
The interest-rate or price trees supported in this toolbox can be either *binomial* (two branches per node) or *trinomial* (three branches per node). Typically, binomial trees assume that underlying interest rates or prices can only either increase or decrease at each node. Trinomial trees allow for a more complex movement of rates or prices. With trinomial trees the movement of rates or prices at each node is unrestricted (for example, up-up-up or unchanged-down-down).

Types of Trees

Financial Derivatives Toolbox trees can be classified as *bushy* or *recombining*. A bushy tree is a tree in which the number of branches increases exponentially relative to observation times; branches never recombine. In this context, 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 tree and a recombining binomial tree are illustrated next.



Bushy Tree



Recombining Binomial Tree

In this toolbox the Heath-Jarrow-Morton model works with bushy trees. The Black-Derman-Toy model, on the other hand, works with recombining binomial trees.

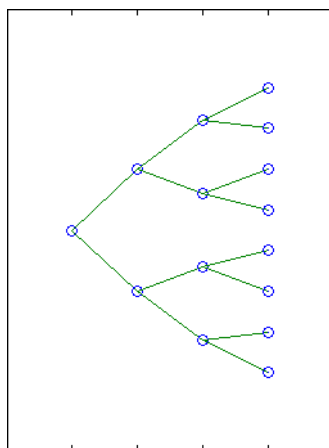
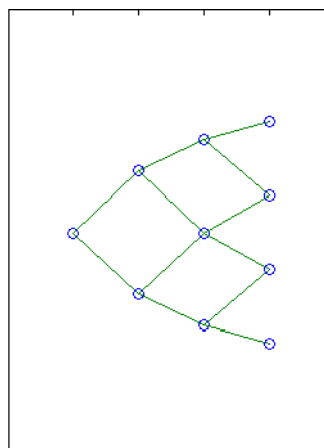
The other two interest rate models supported in this toolbox, Hull-White and Black-Karasinski, work with recombining trinomial trees.

Viewing Rate or Price Movement with This Toolbox

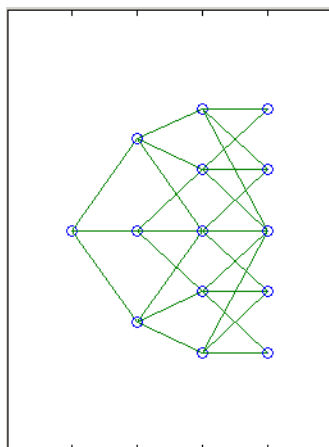
This toolbox provides the data file `deriv.mat` that contains four interest-rate based trees:

- `HJMTree` — A bushy binomial tree
- `BDTTree` — A recombining binomial tree
- `HWTTree` and `BKTree` — Recombining trinomial trees

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)**

With treviewer, you can also see the recombining shape of HW and BK trinomial trees.

**HWTree and BKTree (recombining)**

Hedging

Financial Derivatives Toolbox also includes hedging functionality, allowing the rebalancing of portfolios to reach target costs or target sensitivities, which you can 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 4-2 for a discussion of the hedging process.
- `hedgeopt` for a description of the function that allocates an optimal hedge.
- `hedgeslf` for a description of the function that allocates a self-financing hedge.

Understanding the Interest-Rate Term Structure

- “Interest Rates vs. Discount Factors” on page 2-9
- “Interest-Rate Term Conversions” on page 2-14
- “Functions That Model the Interest-Rate Term Structure” on page 2-18

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 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 factors 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 factors 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 factors.

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

From	To	Rate
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.

Next, 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];
```

Finally, compute the discount factors.

```
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 factors.

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 time based calling syntax is

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

Using this syntax, you 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.

Spot Curve to Forward Curve Conversion

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 forward 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 = 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 forward 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

From	To	Rate
15 Aug 2001	15 Feb 2002	0.0801
15 Feb 2002	15 Aug 2002	0.1155

Alternative Syntax (ratetimes)

The `ratetimes` function can provide the additional output arguments `StartTimes` and `EndTimes`, which represent the time factor equivalents to the `StartDates` and `EndDates` vectors. The `ratetimes` function 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` can also use time factors directly, assuming 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 forward 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
```

Functions That Model the Interest-Rate Term Structure

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 2-19 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 2-21 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 Financial Toolbox documentation.

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 on 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 on 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.

Computing Prices and Sensitivities Using the Interest-Rate Term Structure

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

- “Computing Instrument Prices” on page 2-24
- “Computing Instrument Sensitivities” on page 2-26

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-rate 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.

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 additionally support computations involving options and interest-rate floors and caps. See “Computing Prices and Sensitivities Using Interest-Rate Models” on page 2-49 for information on computing price and sensitivities of financial instruments using the HJM and BDT models.

Computing Instrument Prices

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

<code>bondbyzero:</code>	Price a bond by a set of zero curves
<code>fixedbyzero:</code>	Price a fixed-rate note by a set of zero curves
<code>floatbyzero:</code>	Price a floating-rate note by a set of zero curves
<code>swapbyzero:</code>	Price a swap by a set of zero curves

You can use each of these functions individually to price an instrument. Consult the reference pages for specific information on using these functions.

`intenvprice` takes as input an interest-rate term structure created with `intenvprice`, and a portfolio of interest-rate contingent derivatives instruments created with `instadd`. To learn more about `instadd` and the interest-rate term structure, see Chapter 1, “Getting Started”.

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 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.

Computing Instrument Sensitivities

In general, you can compute sensitivities either as dollar price changes or as percentage price changes. The toolbox reports all sensitivities as dollar sensitivities.

Using the interest-rate term structure, you can calculate two types of derivative price sensitivities, 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 that uses `intenvsens` to calculate both sensitivities and prices.

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

Display the results in a single matrix in bank format.

```
All = [Delta Gamma Price]

All =

           -272.64           1029.84           98.72
```

-347.44	1622.65	97.53
-272.64	1029.84	98.72
-1.04	3.31	100.55
-282.04	1059.62	3.69

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

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

ans =

-2.76	10.43	98.72
-3.56	16.64	97.53
-2.76	10.43	98.72
-0.01	0.03	100.55
-76.39	286.98	3.69

Understanding Interest-Rate Models

Financial Derivatives Toolbox supports the Black-Derman-Toy (BDT), Black-Karasinski (BK), Heath-Jarrow-Morton (HJM), and Hull-White (HW) interest-rate models.

- “Building a Tree of Forward Rates” on page 2-29
- “Specifying the Volatility Model (VolSpec)” on page 2-31
- “Specifying the Interest-Rate Term Structure (RateSpec)” on page 2-34
- “Specifying the Time Structure (TimeSpec)” on page 2-35
- “Examples of Tree Creation” on page 2-36
- “Examining Trees” on page 2-37

The Heath-Jarrow-Morton 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 on a statistical process. For further explanation, see the book *Modelling Fixed Income Securities and Interest Rate Options* by Robert A. Jarrow.

The Black-Derman-Toy model is another analytical model commonly 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.

The Hull-White model incorporates the initial term structure of interest rates and the volatility term structure to build a trinomial recombining tree of short rates. The resulting tree is used to value interest rate dependent securities. The implementation of the Hull-White model in Financial Derivatives Toolbox is limited to one factor.

The Black-Karasinski model is a single factor, log-normal version of the Hull-White model.

For further information on the Hull-White and Black-Karasinski models, see the book *Options, Futures, and Other Derivatives* by John C. Hull.

Building a Tree of Forward Rates

The 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 a forward-rate tree using Financial Derivatives Toolbox.

Note To avoid needless repetition, this document uses the HJM and BDT models to illustrate the creation and use of interest-rate trees. The HW and BK models are similar to the BDT model. Where specific differences exist, they are documented in “HW and BK Tree Structures” on page 2-45.

The MATLAB functions that create rate trees are `hjmtree` and `bdttree`. The `hjmtree` function creates the structure, `HJMTree`, containing time and forward-rate information for a bushy tree. The `bdttree` function creates a similar structure, `BDTTree`, for a recombining tree.

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

These functions take three structures as input arguments:

- The volatility model `VolSpec`. (See “Specifying the Volatility Model (`VolSpec`)” on page 2-31.)
- The interest-rate term structure `RateSpec`. (See “Specifying the Interest-Rate Term Structure (`RateSpec`)” on page 2-34.)
- The tree time layout `TimeSpec`. (See “Specifying the Time Structure (`TimeSpec`)” on page 2-35.)

An easy way to visualize any trees you create is with the `treeviewer` function, which displays trees in a graphical manner. See “Graphical Representation of Trees” on page 2-61 for information about `treeviewer`.

Calling Sequence

The calling syntax for `hjmtree` is

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

Similarly, the calling syntax for `bdttree` is

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

Each of these functions requires `VolSpec`, `RateSpec`, and `TimeSpec` input arguments:

- `VolSpec` is a structure that specifies the forward-rate volatility process. You create `VolSpec` using either of the functions `hjmvolspec` or `bdtvolspec`.

The `hjmvolspec` function supports the specification of up to three factors. It handles these models for the volatility of the interest-rate term structure:

- Constant
- Stationary
- Exponential
- Vasicek
- Proportional

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

The `bdtvolspec` function supports only a single volatility factor. The volatility remains constant between pairs of nodes on the tree. You supply the input volatility values in a vector of decimal values. See `bdtvolspec` for details.

- `RateSpec` is the interest-rate specification of the initial rate curve. You create this structure with the function `intenvset`. (See “Functions That Model the Interest-Rate Term Structure” on page 2-18.)
- `TimeSpec` is the tree time layout specification. You create this variable with the functions `hjmtimespec` or `bdttimespec`. It represents the mapping

between level times and level dates for rate quoting. This structure indirectly determines the number of levels in the tree.

Specifying the Volatility Model (VolSpec)

Because HJM supports multifactor (up to three) volatility models while BDT (also, BK and HW) supports only a single volatility factor, the `hjmvolspec` and `bdtvolspec` functions require somewhat different inputs and generate slightly different outputs. For examples, see “Creating an HJM Volatility Model” on page 2-31. For BDT examples see “Creating a BDT Volatility Model” on page 2-32.

Creating an HJM Volatility Model

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 capital 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 function that creates it. Each factor specification starts with a string specifying the name of the factor, followed by the pertinent parameters.

HJM Volatility Specification Example. 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.

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

```
HJMVolSpec =
    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, VolSpec.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
HJMVolSpec = hjmvolspec('Proportional', CurveProp, CurveTerm,...
1e6,'Exponential', Sigma_0, Lambda)

HJMVolSpec =

    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.

Creating a BDT Volatility Model

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

```
BDTVolSpec = 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`.

BDT Volatility Specification Example. 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 forward-rate specification from which the tree rates are derived. “Functions That Model the Interest-Rate Term Structure” on page 2-18 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.

Rate Specification Creation Example

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 interest-rate tree. This structure defines the mapping between the observation times at each level of the tree and the corresponding dates.

`TimeSpec` is built using either the `hjmtimespec` or `bdttimespec` function. These functions require three input arguments:

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

For example, 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.

Creating a Time Specification

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

HJM Time Specification Example. Consider the example

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

HJMTimeSpec =

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

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

Creating a BDT Time Specification. Consider the example

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

BDTTimeSpec =

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

Examples of Tree Creation

Use the `VolSpec`, `RateSpec`, and `TimeSpec` you have previously created as inputs to the functions used to create HJM and BDT trees.

Creating an HJM Tree

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

HJMTree = hjmtree(HJMVolSpec, RateSpec, HJMTimeSpec)

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]}
```

Creating a BDT Tree

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

```
BDTTree = bdttree(BDTVVolSpec, RateSpec, BDTTimeSpec)

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]}
```

Examining Trees

When working with the models, Financial Derivatives Toolbox uses trees to represent forward rates, prices, and so on. 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	Attributes
BDTInstSet	1x1	15956	struct	
BDTTree	1x1	5138	struct	
BKInstSet	1x1	15946	struct	
BKTree	1x1	5904	struct	
CRRInstSet	1x1	12434	struct	
CRRTree	1x1	5058	struct	
EQPInstSet	1x1	12434	struct	
EQPTree	1x1	5058	struct	
HJMInstSet	1x1	15948	struct	
HJMTree	1x1	5838	struct	
HWInstSet	1x1	15946	struct	
HWTree	1x1	5904	struct	
ITTInstSet	1x1	12438	struct	
ITTree	1x1	8862	struct	
ZeroInstSet	1x1	10282	struct	
ZeroRateSpec	1x1	1580	struct	

HJM Tree Structure

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

```
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. MATLAB represents it 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 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, $t_{Obs} = 2$. 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 Financial Derivatives Toolbox, such as forward-rate trees or price trees.

BDT Tree Structure

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}
```

`FwdTree` contains the actual rate tree. **MATLAB** represents it 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
```

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, $t_{Obs} = 1$. 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 branch.

```
BDTTree.FwdTree{3}
```

```
ans =
```

```
1.0976    1.1377    1.1942
```

The third node represents the second observation time, $t_{Obs} = 2$. 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, $t_{Obs} = 3$. 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).

Isolating a Specific Node. 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(BDTree.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.

HW and BK Tree Structures

The HW and BK tree structures are very similar to the BDT tree structure. You can see this if you examine the sample HW tree contained in the file `deriv.mat`.

```
load deriv.mat:
```

```
HWTree
```

```
FinObj: 'HWFwdTree'
VolSpec: [1x1 struct]
TimeSpec: [1x1 struct]
RateSpec: [1x1 struct]
tObs: [0 1 2 3]
dObs: [731947 732313 732678 733043]
CFlowT: {[4x1 double] [3x1 double] [2x1 double] [4]}
Probs: {[3x1 double] [3x3 double] [3x5 double]}
Connect: {[2] [2 3 4] [2 2 3 4 4]}
FwdTree: {1x4 cell}
```

All fields of this structure are similar to their BDT counterparts. There are two additional fields not present in BDT: Probs and Connect. The Probs field represents the occurrence probabilities at each branch of each node in the tree. The Connect field describes the connectivity of the nodes of a given tree level to nodes to the next tree level.

Probs Field. While BDT and one-factor HJM models have equal probabilities for each branch at a node, HW and BK do not. For HW and BK trees, the Probs field indicates the likelihood that a particular branch will be taken in moving from one node to another node on the next level.

The Probs field consists of a cell array with one cell per tree level. Each cell is a 3-by-NUMNODES array with the top row representing the probability of an up movement, the middle row representing the probability of a middle movement, and the last row the probability of a down movement.

As an illustration, consider the first two elements of the Probs field of the structure, corresponding to the first (root) and second levels of the tree.

```
HWTree.Probs{1}
```

```
0.16666666666667
0.66666666666667
0.16666666666667
```

```
HWTree.Probs{2}
```

```
0.12361333418768  0.16666666666667  0.21877591615172
0.65761074966060  0.66666666666667  0.65761074966060
0.21877591615172  0.16666666666667  0.12361333418768
```

Reading from top to bottom, the values in `HWTree.Probs{1}` correspond to the up, middle, and down probabilities at the root node.

`HWTree.Probs{2}` is a 3-by-3 matrix of values. The first column represents the top node, the second column represents the middle node, and the last column represents the bottom node. As with the root node, the first, second, and third rows hold the values for up, middle, and down branching off each node.

As expected, the sum of all the probabilities at any node equals 1.

```
sum(HWTree.Probs{2})

1.0000    1.0000    1.0000
```

Connect Field. The other field that distinguishes HW and BK tree structures from the BDT tree structure is `Connect`. This field describes how each node in a given level connects to the nodes of the next level. The need for this field arises from the possibility of nonstandard branching in a tree.

The `Connect` field of the HW tree structure consists of a cell array with one cell per tree level.

```
HWTree.Connect

ans =

    [2]    [1x3 double]    [1x5 double]
```

Each cell contains a 1-by-`NUMNODES` vector. Each value in the vector relates to a node in the corresponding tree level and represents the index of the node in the next tree level that the middle branch of the node connects to.

If you subtract 1 from the values contained in `Connect`, you reveal the index of the nodes in the next level that the up branch connects to. If you add 1 to the values, you reveal the index of the corresponding down branch.

As an illustration, consider `HWTree.Connect{1}`:

```
HWTree.Connect{1}

ans =

    2
```

This indicates that the middle branch of the root node connects to the second (from the top) node of the next level, as expected. If you subtract 1 from this value, you obtain 1, which tells you that the up branch goes to the top

node. If you add 1, you obtain 3, which points to the last node of the second level of the tree.

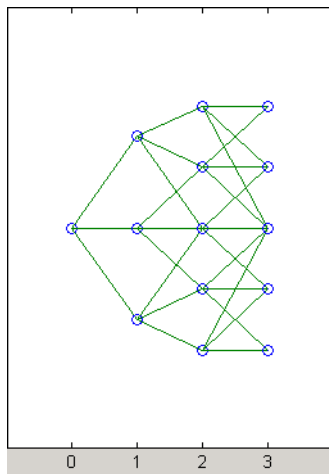
Now consider level 3 in this example:

```
HWTree.Connect{3}
```

2 2 3 4 4

On this level, there is nonstandard branching. This can be easily recognized because the middle branch of two nodes is connected to the same node on the next level.

To visualize this, consider the following illustration of the tree.



Here it becomes apparent that there is nonstandard branching at the third level of the tree, on the top and bottom nodes. The first and second nodes connect to the same trio of nodes on the next level. Similar branching occurs at the bottom and next-to-bottom nodes of the tree.

Computing Prices and Sensitivities Using Interest-Rate Models

This section explains how to use Financial Derivatives Toolbox to compute prices and sensitivities of several financial instruments. See

- “Computing Instrument Prices” on page 2-49
- “Computing Instrument Sensitivities” on page 2-58

Note For purposes of illustration, this section relies on the HJM and BDT models. The HW and BK functions that perform price and sensitivity computations are not explicitly shown here. Functions that use the HW and BK models operate similarly to the BDT model.

Computing Instrument Prices

The portfolio pricing functions `hjmprice` and `bdtpprice` calculate the price of any set of supported instruments, based on an interest-rate tree. The functions are capable of pricing these instrument types:

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

For example, the syntax for calling `hjmprice` is

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

Similarly, the calling syntax for `bdtpprice` is

```
[Price, PriceTree] = bdtpprice(BDTree, InstSet, Options)
```

Each function requires two input arguments: the interest-rate tree and the set of instruments, `InstSet`. An optional argument `Options` further controls the pricing and the output displayed. (See Appendix A, “Derivatives Pricing Options” for information about the `Options` argument.)

`HJMTree` is the Heath-Jarrow-Morton tree sampling of a forward-rate process, created using `hjmtree`. `BDTree` is the Black-Derman-Toy tree sampling of an interest-rate process, created using `bdttree`. See “Building a Tree of Forward Rates” on page 2-29 to learn how to create these structures.

`InstSet` is the set of instruments to be priced. This structure represents the set of instruments to be priced independently using the model. Chapter 1, “Getting Started” explains how to create this variable.

`Options` is an options structure created with the function `derivset`. This structure defines how the 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 the pricing function, a default `Options` structure is used. The pricing options structure is described in “Pricing Options Structure” on page A-2.

The portfolio pricing functions classify the instruments and call the appropriate instrument-specific pricing function for each of the instrument types. The HJM instrument-specific pricing functions are `bondbyhjm`, `cfbyhjm`, `fixedbyhjm`, `floatbyhjm`, `optbndbyhjm`, and `swapbyhjm`. A similarly named set of functions exists for BDT models. For a list of these, see “Price and Sensitivity from Black-Derman-Toy Trees” on page 5-3.

You can also use these functions directly to calculate the price of sets of instruments of the same type. See Chapter 6, “Functions — Alphabetical List” for these individual functions for further information.

HJM Pricing Example

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	Attributes
BDTInstSet	1x1	15956	struct	
BDTTree	1x1	5138	struct	
BKInstSet	1x1	15946	struct	
BKTree	1x1	5904	struct	
CRRInstSet	1x1	12434	struct	
CRRTree	1x1	5058	struct	
EQPInstSet	1x1	12434	struct	
EQPTree	1x1	5058	struct	
HJMInstSet	1x1	15948	struct	
HJMTree	1x1	5838	struct	
HWInstSet	1x1	15946	struct	
HWTree	1x1	5904	struct	
ITTInstSet	1x1	12438	struct	
ITTTree	1x1	8862	struct	
ZeroInstSet	1x1	10282	struct	
ZeroRateSpec	1x1	1580	struct	

`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.

BDT Pricing Example

Load the MAT-file `deriv.mat` 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	Attributes
BDTInstSet	1x1	15956	struct	
BDTTree	1x1	5138	struct	
BKInstSet	1x1	15946	struct	
BKTree	1x1	5904	struct	
CRRInstSet	1x1	12434	struct	
CRRTree	1x1	5058	struct	
EQPInstSet	1x1	12434	struct	
EQPTree	1x1	5058	struct	
HJMInstSet	1x1	15948	struct	
HJMTree	1x1	5838	struct	
HWInstSet	1x1	15946	struct	
HWTree	1x1	5904	struct	
ITTInstSet	1x1	12438	struct	

```

ITTree          1x1          8862  struct
ZeroInstSet     1x1          10282 struct
ZeroRateSpec    1x1          1580  struct

```

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 `bdtpprice`.

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

```
Price = bdtpprice(BDTree, BDTInstSet)
Warning: Not all cash flows are aligned with the tree. Result will
be approximated.

Price =

    95.5030
    93.9079
     1.7657
    95.5030
   100.4865
     1.4863
     0.0245
     7.4222
```

Price Vector Output

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 the HJM example, the prices in the `Price` vector correspond to the instruments in this 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).

In the BDT example, the prices in the `Price` vector correspond to the instruments in this 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 Output

If you call a pricing function with two output arguments, for example,

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

you generate a price tree along with the price information.

The optional output price tree structure `PriceTree` holds all the pricing information.

HJM Price Tree. In the HJM example, 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 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*.

BDT Price Tree. The BDT 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 field, `tObs`, represents the observation time of each level of `Ptree` and `AITree`, with units in terms of compounding periods.

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.

```
[Price, PriceTree] = bdtprice(BDTree, BDTInstSet)
```

```
PriceTree.PTree{1}
```

```
ans =
```

```
95.5030  
93.9079  
1.7657  
95.5030  
100.4865  
1.4863  
0.0245  
7.4222
```

Computing Instrument Sensitivities

The toolbox reports sensitivities either as dollar price changes or percentage price changes. The delta, gamma, and vega sensitivities that the toolbox computes are dollar sensitivities.

The functions `hjsens` and `bdtens` compute the delta, gamma, and vega sensitivities of instruments using an interest-rate tree. They also optionally return the calculated price for each instrument. The sensitivity functions require the same two input arguments used by the pricing functions (`HJMTree` and `HJMInstSet` for HJM; `BDTree` and `BDTInstSet` for BDT).

Sensitivity functions calculate 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.

HJM Sensitivities Example

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.

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 `HJMIInstSet`. To view the *per-dollar sensitivities*, divide each dollar sensitivity by the corresponding instrument price.

BDT Sensitivities Example

The calling syntax for the function is

```
[Delta, Gamma, Vega, Price] = bdt sens(BDTTree, BDTInstSet);
```

Arrange the sensitivities and prices into a single matrix.

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

```
All =
```

-232.67	803.71	-0.00	95.50
-281.05	1181.93	-0.01	93.91
-50.54	246.02	5.31	1.77
-232.67	803.71	0	95.50
0.84	2.45	0	100.49
78.38	748.98	13.54	1.49
-4.36	382.06	2.50	0.02
-253.23	863.81	0	7.42

To view the *per-dollar sensitivities*, divide each dollar sensitivity by the corresponding instrument price.

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

```
All =
```

-2.44	8.42	-0.00	95.50
-2.99	12.59	-0.00	93.91
-28.63	139.34	3.01	1.77
-2.44	8.42	0	95.50
0.01	0.02	0	100.49
52.73	503.92	9.11	1.49
-177.89	15577.42	101.87	0.02
-34.12	116.38	0	7.42

Graphical Representation of Trees

- “Observing Interest Rates” on page 2-61
- “Observing Instrument Prices” on page 2-65

You can use the function `treeviewer` to display a graphical representation of a tree, allowing you to examine interactively the prices and rates on the nodes of the tree until maturity. To get started with this process, first load the data file `deriv.mat` included in this toolbox.

```
load deriv.mat
```

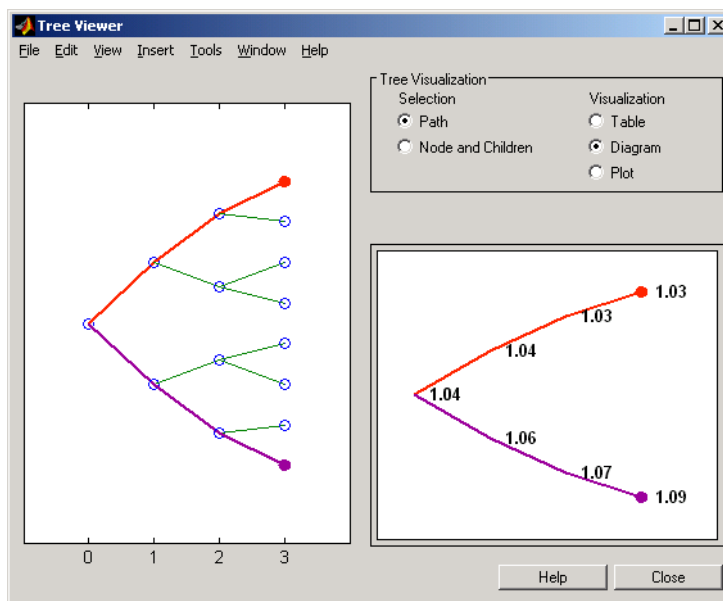
Note `treeviewer` price tree diagrams follow the convention that increasing prices appear on the upper branch of a tree and, consequently, decreasing prices appear on the lower branch. Conversely, for interest rate displays, *decreasing* interest rates appear on the upper branch (prices are rising) and *increasing* interest rates on the lower branch (prices are falling).

For information on the use of `treeviewer` to observe interest rate movement, see “Observing Interest Rates” on page 2-61. For information on using `treeviewer` to observe the movement of prices, see “Observing Instrument Prices” on page 2-65.

Observing Interest Rates

If you provide the name of an interest rate tree to the `treeviewer` function, it displays a graphical view of the path of interest rates. For example, here is the `treeviewer` representation of all the rates along both the up and down branches of `HJMTree`.

```
treeviewer(HJMTree)
```



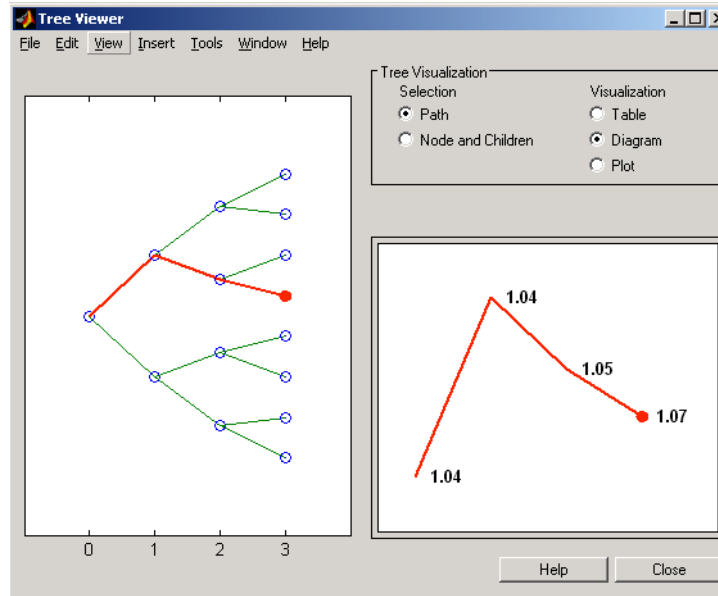
The example in “Isolating a Specific Node for a CRRTree” on page 3-19 used `bushpath` to find the path of forward rates along an HJM tree by taking the first branch up and then two branches down the rate tree.

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

```
FRates =
```

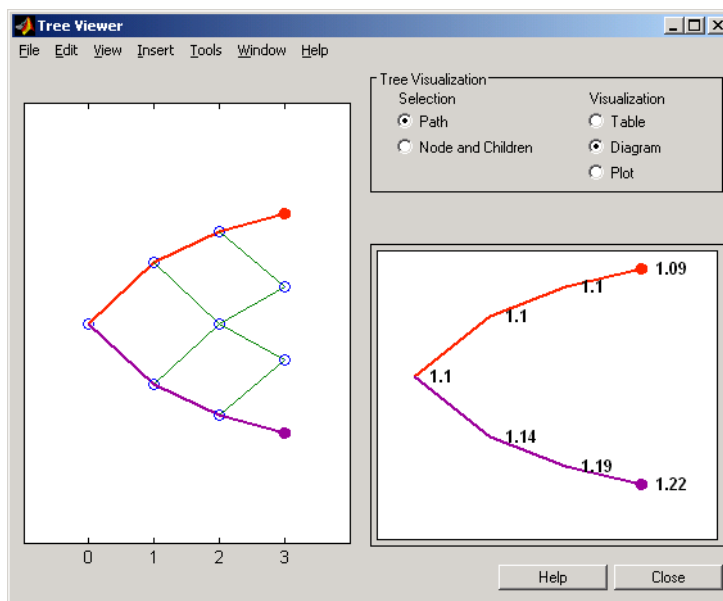
```
1.0356
1.0364
1.0526
1.0674
```

With the `treeviewer` function you can display the identical information by clicking along the same sequence of nodes, as shown next.



Next is a treeviewer representation of interest rates along several branches of BDTTree.

```
treeviewer(BDTTree)
```



Note When using `treeviewer` with recombining trees, such as BDT, BK, and HW, you must click each node in succession from the beginning to the end. Because these trees can recombine, `treeviewer` is unable to complete the path automatically.

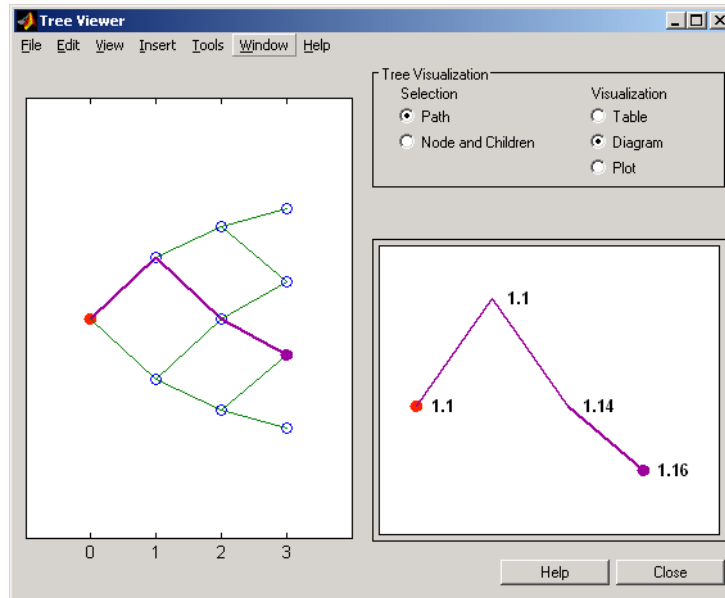
The example in “Isolating a Specific Node for a CRRTree” on page 3-19 used `treepath` to find the path of interest rates taking the first branch up and then two branches down the rate tree.

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

```
FRates =
```

```
1.1000
1.0979
1.1377
1.1606
```

You can display the identical information by clicking along the same sequence of nodes, as shown next.

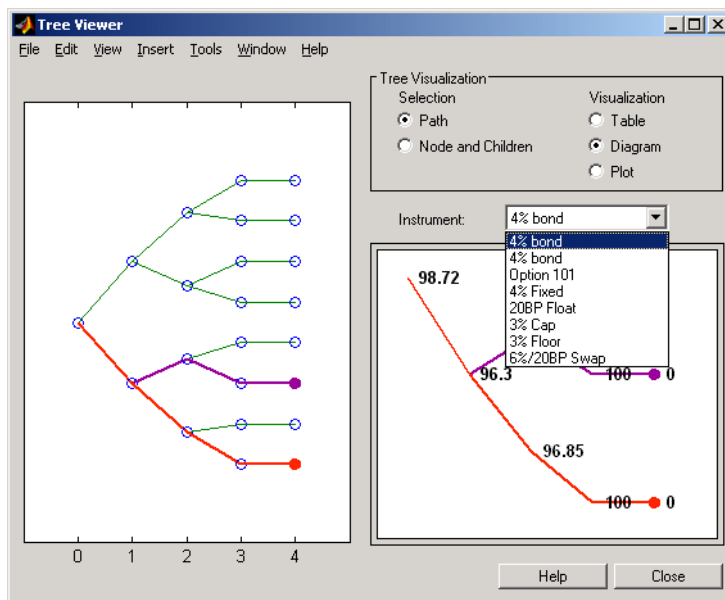


Observing Instrument Prices

To use `treeviewer` to display a tree of instrument prices, provide the name of an instrument set along with the name of a price tree in your call to `treeviewer`, for example:

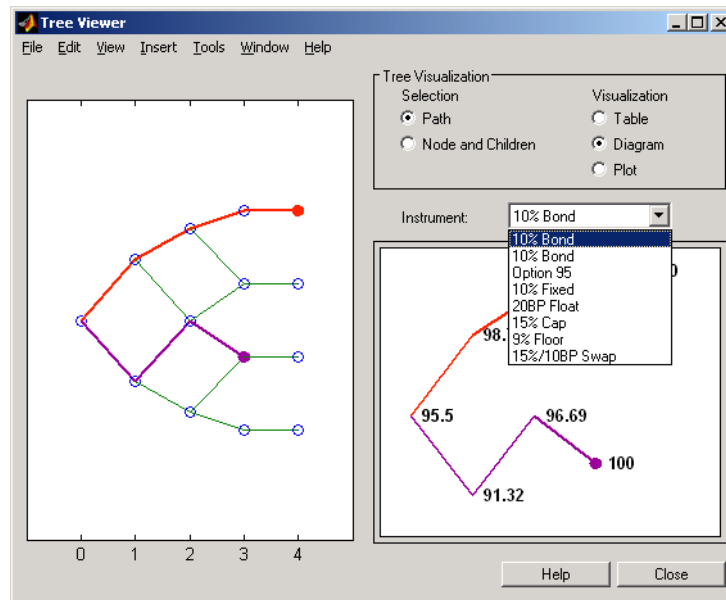
```
load deriv.mat
[Price, PriceTree] = hjmprice(HJMTTree, HJMInstSet);
treeviewer(PriceTree, HJMInstSet)
```

With treeviewer you select *each instrument individually* in the instrument portfolio for display.



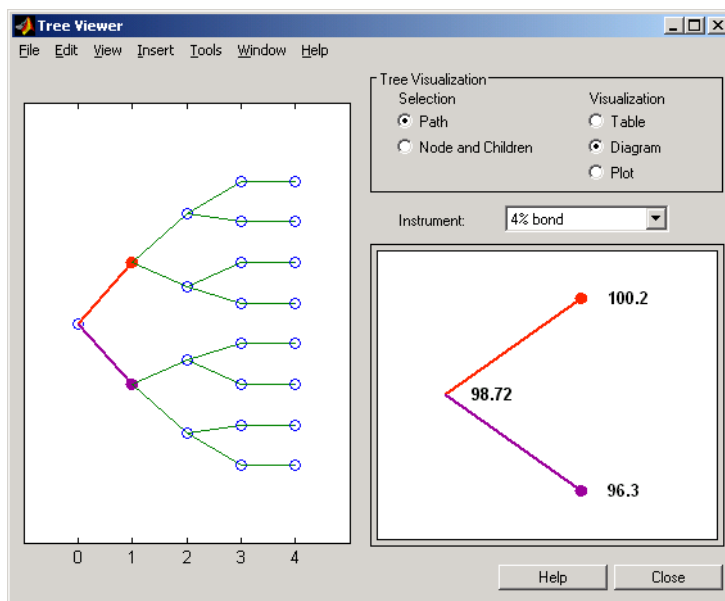
You can use an analogous process to view instrument prices based on the BDT interest rate tree included in `deriv.mat`.

```
load deriv.mat
[BDTPrice, BDTPriceTree] = bdtprice(BDTTree, BDTInstSet);
treeviewer(BDTPriceTree, BDTInstSet)
```

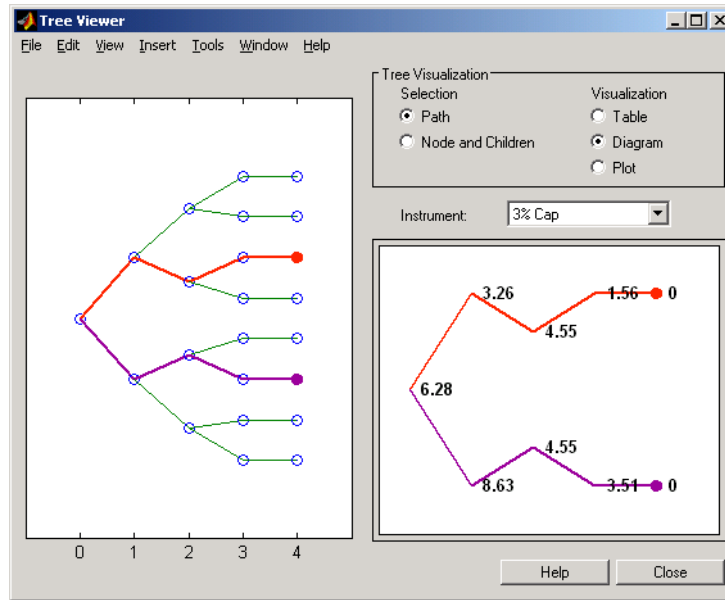


Valuation Date Prices

You can use treeviewer instrument-by-instrument to observe instrument prices through time. For the first 4% bond in the HJM 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.



Additional Observation Times

The second node represents the first rate observation time, $t_{0bs} = 1$. This node displays two states, one representing the branch going up and the other one representing the branch going down.

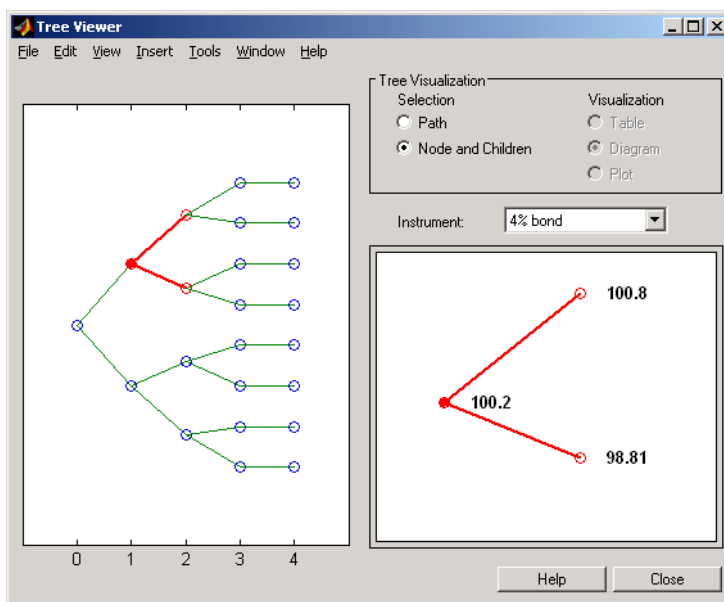
Examine the prices of the node corresponding to the up branch.

```
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)
```

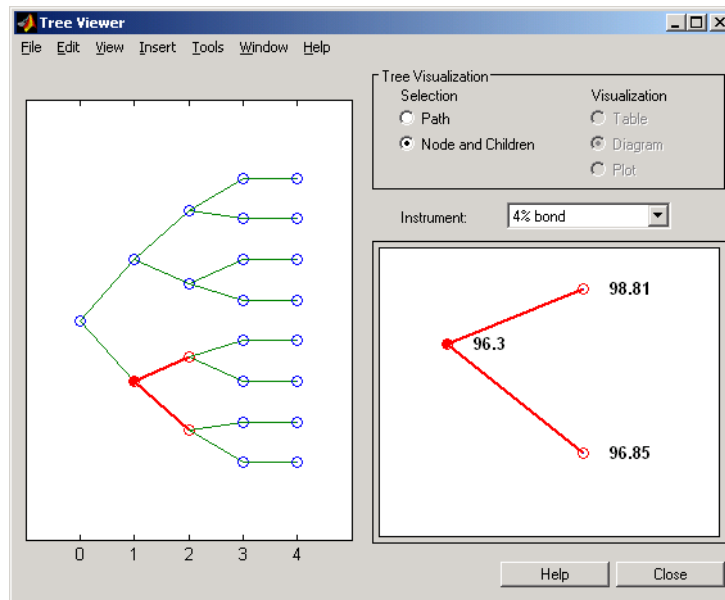
ans =

```

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.



Equity Derivatives

Understanding Equity Trees (p. 3-2)	CRR, EQP, and ITT equity pricing models.
Understanding Equity Exotic Options (p. 3-22)	Asian, barrier, compound, and lookback exotic options. Also, American, European, and Bermuda put and call options on equities.
Computing Prices and Sensitivities for Equity Derivatives (p. 3-26)	Use of <code>crrprice</code> , <code>eqpprice</code> , and <code>ittprice</code> to compute the prices of vanilla options and exotic options.

Understanding Equity Trees

- “Building Equity Binary Trees” on page 3-3
- “Building Implied Trinomial Trees” on page 3-8
- “Examining Equity Trees ” on page 3-16
- “Differences Between CRR and EQP Tree Structures” on page 3-20

Financial Derivatives Toolbox supports three types of recombining tree models to represent the evolution of stock prices:

- Cox-Ross-Rubinstein (CRR) model
- Equal probabilities (EQP) model
- Implied trinomial tree (ITT) model

For a discussion of recombining trees, see “Rate and Price Trees” on page 2-5.

The CRR, EQP, and ITT models are examples of discrete time models. A discrete time model divides time into discrete bits; prices can only be computed at these specific times.

The CRR model is one of the most common methods used to model the evolution of stock processes. The strength of the CRR model lies in its simplicity. It is a good model when dealing with a large number of tree levels. The CRR model yields the correct expected value for each node of the tree and provides a good approximation for the corresponding local volatility. The approximation becomes better as the number of time steps represented in the tree is increased.

The EQP model is another discrete time model. It has the advantage of building a tree with the exact volatility in each tree node, even with small numbers of time steps. It also provides better results than CRR in some given trading environments, for example, when stock volatility is low and interest rates are high. However, this additional precision causes increased complexity, which is reflected in the number of calculations required to build a tree.

The ITT model is a CRR-style implied trinomial tree which takes advantage of prices quoted from liquid options in the market with varying strikes and

maturities to build a tree that more accurately represents the market. An ITT model is commonly used to price exotic options in such a way that they are consistent with the market prices of standard options.

Building Equity Binary Trees

The tree of stock prices is the fundamental unit representing the evolution of the price of a stock over a given period of time. The MATLAB functions `crrtree` and `eqptree` create CRR trees and EQP trees respectively. These functions create an output tree structure along with information about the parameters used for creating the tree.

The functions `crrtree` and `eqptree` take three structures as input arguments:

- The stock parameter structure `StockSpec`
- The interest-rate term structure `RateSpec`
- The tree time layout structure `TimeSpec`

Calling Sequence for Equity Binary Trees

The calling syntax for `crrtree` is:

```
CRRTree = crrtree (StockSpec, RateSpec, TimeSpec)
```

Similarly, the calling syntax for `eqptree` is:

```
EQPTree = eqptree (StockSpec, RateSpec, TimeSpec)
```

Both functions require the structures `StockSpec`, `RateSpec`, and `TimeSpec` as input arguments:

- `StockSpec` is a structure that specifies parameters of the stock whose price evolution is represented by the tree. This structure, created using the function `stockspec`, contains information such as the stock's original price, its volatility, and its dividend payment information.
- `RateSpec` is the interest-rate specification of the initial rate curve. Create this structure with the function `intenvset`.
- `TimeSpec` is the tree time layout specification. Create these structures with the functions `crrtimespec` and `eqptimespec`. The structures contain

information regarding the mapping of relevant dates into the tree structure, plus the number of time steps used for building the tree.

Specifying the Stock Structure for Equity Binary Trees

The structure `StockSpec` encapsulates the stock-specific information required for building the binary tree of an individual stock's price movement.

You generate `StockSpec` with the function `stockspec`. This function requires two input arguments and accepts up to three additional input arguments that depend on the existence and type of dividend payments.

The syntax for calling `stockspec` is:

```
StockSpec = stockspec(Sigma, AssetPrice, DividendType, ...  
DividendAmounts, ExDividendDates)
```

where:

- `Sigma` is the decimal annual volatility of the underlying security.
- `AssetPrice` is the price of the stock at the valuation date.
- `DividendType` is a string specifying the type of dividend paid by the stock. Allowed values are `cash`, `constant`, or `continuous`.
- `DividendAmounts` has a value that depends on the specification of `DividendType`. For `DividendType cash`, `DividendAmounts` is a vector of cash dividends. For `DividendType constant`, it is a vector of constant annualized dividend yields. For `DividendType continuous`, it is a scalar representing a continuously annualized dividend yield.
- `ExDividendDates` also has a value that depends on the nature of `DividendType`. For `DividendType cash` or `constant`, `ExDividendDates` is vector of dividend dates. For `DividendType continuous`, `ExDividendDates` is ignored.

Stock Structure Example Using a Binary Tree

Consider a stock with a price of \$100 and an annual volatility of 15%. Assume that the stock pays three cash \$5.00 dividends on dates January 01, 2003; July 01, 2003; and January 01, 2004. You specify these parameters in MATLAB as:


```

Sigma = 0.15;
AssetPrice = 100;
DividendType = 'cash';
DividendAmounts = [5; 5; 5];
ExDividendDates = {'jan-01-2004', 'july-01-2005', 'jan-01-2006'};

StockSpec = stockspec(Sigma, AssetPrice, DividendType, ...
    DividendAmounts, ExDividendDates)

StockSpec =

    FinObj: 'StockSpec'
    Sigma: 0.1500
    AssetPrice: 100
    DividendType: 'cash'
    DividendAmounts: [3x1 double]
    ExDividendDates: [3x1 double]

```

Specifying the Interest-Rate Term Structure for Equity Binary Trees

The `RateSpec` structure defines the interest rate environment used when building the stock price binary tree. “Functions That Model the Interest-Rate Term Structure” on page 2-18 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.

Specifying the Tree-Time Term Structure for Equity Binary Trees

The `TimeSpec` structure defines the tree layout of the binary tree:

- It maps the valuation and maturity dates to their corresponding times.
- It defines the time of the levels of the tree by dividing the time span between valuation and maturity into equally spaced intervals. By specifying the number of intervals, you define the granularity of the tree time structure.

The syntax for building a `TimeSpec` structure is:

```

TimeSpec = crtimespec(ValuationDate, Maturity, NumPeriods)
TimeSpec = eqptimespec(ValuationDate, Maturity, NumPeriods)

```

where:

- `ValuationDate` is a scalar date marking the pricing date and first observation in the tree (location of the root node). You enter `ValuationDate` either as a serial date number (generated with `datenum`) or a date string.
- `Maturity` is a scalar date marking the maturity of the tree, entered as a serial date number or a date string.
- `NumPeriods` is a scalar defining the number of time steps in the tree; for example, `NumPeriods = 10` implies 10 time steps and 11 tree levels (0, 1, 2, ..., 9, 10).

TimeSpec Example Using a Binary Tree

Consider building a CRR tree, with a valuation date of January 1, 2003, a maturity date of January 1, 2008, and 20 time steps. You specify these parameters in MATLAB as:

```
ValuationDate = 'Jan-1-2003';  
Maturity = 'Jan-1-2008';  
NumPeriods = 20;  
TimeSpec = crrtimespec(ValuationDate, Maturity, NumPeriods)  
TimeSpec =
```

```
    FinObj: 'BinTimeSpec'  
ValuationDate: 731582  
    Maturity: 733408  
    NumPeriods: 20  
        Basis: 0  
    EndMonthRule: 1  
        tObs: [1x21 double]  
        dObs: [1x21 double]
```

Two vector fields in the `TimeSpec` structure are of particular interest: `dObs` and `tObs`. These two fields represent the observation times and corresponding dates of all tree levels, with `dObs(1)` and `tObs(1)`, respectively, representing the root node (`ValuationDate`), and `dObs(end)` and `tObs(end)` representing the last tree level (`Maturity`).

Note There is no relationship between the dates specified for the tree and the implied tree level times, and the maturities specified in the interest rate term structure. The rates in RateSpec are interpolated or extrapolated as needed to meet the time distribution of the tree.

Examples of Binary Tree Creation

You can now use the StockSpec and TimeSpec structures described previously to build an equal probability tree (EQPTree) and a CRR tree (CRRTree). First, you must define the interest rate term structure. For this example, assume that the interest rate is fixed at 10% annually between the valuation date of the tree (January 1, 2003) until its maturity.

```
ValuationDate = 'Jan-1-2003';
Maturity = 'Jan-1-2008';
Rate = 0.1;
RateSpec = intenvset('Rates', Rate, 'StartDates', ...
    ValuationDate, 'EndDates', Maturity, 'Compounding', -1);
```

To build a CRRTree, enter:

```
CRRTree = crrtree(StockSpec, RateSpec, TimeSpec)

CRRTree =

    FinObj: 'BinStockTree'
    Method: 'CRR'
    StockSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
    tObs: [1x21 double]
    dObs: [1x21 double]
    STree: {1x21 cell}
    UpProbs: [1x20 double]
```

To build an EQPTree, enter:

```
EQPTree = eqptree(StockSpec, RateSpec, TimeSpec)
```

```
EQPTree =  
  
    FinObj: 'BinStockTree'  
    Method: 'EQP'  
    StockSpec: [1x1 struct]  
    TimeSpec: [1x1 struct]  
    RateSpec: [1x1 struct]  
        tObs: [1x21 double]  
        dObs: [1x21 double]  
    STree: {1x21 cell}  
    UpProbs: [1x20 double]
```

Building Implied Trinomial Trees

The tree of stock prices is the fundamental unit representing the evolution of the price of a stock over a given period of time. The MATLAB function `itttree` creates an output tree structure along with the information about the parameters used to create the tree.

The function `itttree` takes four structures as input arguments:

- The stock parameter structure `StockSpec`
- The interest-rate term structure `RateSpec`
- The tree time layout structure `TimeSpec`
- The stock option specification structure `StockOptSpec`

Calling Sequence for Implied Trinomial Trees

The calling syntax for `itttree` is:

```
ITTree = itttree (StockSpec,RateSpec,TimeSpec,StockOptSpec)
```

- `StockSpec` is a structure that specifies parameters of the stock whose price evolution is represented by the tree. This structure, created using the function `stockspec`, contains information such as the stock's original price, its volatility, and its dividend payment information.
- `RateSpec` is the interest-rate specification of the initial rate curve. Create this structure with the function `intenvset`.

- `TimeSpec` is the tree time layout specification. Create these structures with the function `itttimespec`. This structure contains information regarding the mapping of relevant dates into the tree structure, plus the number of time steps used for building the tree.
- `StockOptSpec` is a structure containing parameters of European stock options instruments. Create this structure with the function `stockoptspec`.

Specifying the Stock Structure for Implied Trinomial Trees

The structure `StockSpec` encapsulates the stock-specific information required for building the trinomial tree of an individual stock's price movement.

You generate `StockSpec` with the function `stockspec`. This function requires two input arguments and accepts up to three additional input arguments that depend on the existence and type of dividend payments.

The syntax for calling `stockspec` is:

```
StockSpec = stockspec(Sigma, AssetPrice, DividendType, ...
    DividendAmounts, ExDividendDates)
```

where:

- `Sigma` is the decimal annual volatility of the underlying security.
- `AssetPrice` is the price of the stock at the valuation date.
- `DividendType` is a string specifying the type of dividend paid by the stock. Allowed values are `cash`, `constant`, or `continuous`.
- `DividendAmounts` has a value that depends on the specification of `DividendType`. For `DividendType` `cash`, `DividendAmounts` is a vector of cash dividends. For `DividendType` `constant`, it is a vector of constant annualized dividend yields. For `DividendType` `continuous`, it is a scalar representing a continuously annualized dividend yield.
- `ExDividendDates` also has a value that depends on the nature of `DividendType`. For `DividendType` `cash` or `constant`, `ExDividendDates` is vector of dividend dates. For `DividendType` `continuous`, `ExDividendDates` is ignored.

Stock Structure Example Using an Implied Trinomial Tree

Consider a stock with a price of \$100 and an annual volatility of 12%. Assume that the stock is expected to pay a dividend yield of 6%. You specify these parameters in MATLAB as:

```
So=100;
DividendYield = 0.06;
Sigma=.12;

StockSpec = stockspec(Sigma, So, 'continuous', DividendYield)

StockSpec =

    FinObj: 'StockSpec'
    Sigma: 0.1200
    AssetPrice: 100
    DividendType: 'continuous'
    DividendAmounts: 0.0600
    ExDividendDates: []
```

Specifying the Interest-Rate Term Structure for Implied Trinomial Trees

The structure `RateSpec` defines the interest rate environment used when building the stock price binary tree. “Functions That Model the Interest-Rate Term Structure” on page 2-18 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.

Specifying the Tree Time Term Structure for Implied Trinomial Trees

The `TimeSpec` structure defines the tree layout of the trinomial tree:

- It maps the valuation and maturity dates to their corresponding times.
- It defines the time of the levels of the tree by dividing the time span between valuation and maturity into equally spaced intervals. By specifying the number of intervals, you define the granularity of the tree time structure.

The syntax for building a `TimeSpec` structure is:

```
TimeSpec = itttimespec(ValuationDate, Maturity, NumPeriods)
```

where:

- ValuationDate is a scalar date marking the pricing date and first observation in the tree (location of the root node). You enter ValuationDate either as a serial date number (generated with datenum) or a date string.
- Maturity is a scalar date marking the maturity of the tree, entered as a serial date number or a date string.
- NumPeriods is a scalar defining the number of time steps in the tree; for example, NumPeriods = 10 implies 10 time steps and 11 tree levels (0, 1, 2, ..., 9, 10).

TimeSpec Example Using an Implied Trinomial Tree

Consider building an ITT tree, with a valuation date of January 1, 2006, a maturity date of January 1, 2008, and four time steps. You specify these parameters in MATLAB as:

```
ValuationDate = '01-01-2006';
EndDate = '01-01-2008';
NumPeriods = 4;

TimeSpec = itttimespec(ValuationDate, EndDate, NumPeriods)

TimeSpec =

    FinObj: 'ITTimeSpec'
ValuationDate: 732678
    Maturity: 733408
    NumPeriods: 4
    Basis: 0
EndMonthRule: 1
    tObs: [0 0.5000 1 1.5000 2]
    dObs: [732678 732860 733043 733225 733408]
```

Two vector fields in the TimeSpec structure are of particular interest: dObs and tObs. These two fields represent the observation times and corresponding dates of all tree levels, with dObs(1) and tObs(1), respectively, representing

the root node (ValuationDate), and dObs(end) and tObs(end) representing the last tree level (Maturity).

Specifying the Option Stock Structure for Implied Trinomial Trees

The StockOptSpec structure encapsulates the option-stock-specific information required for building the implied trinomial tree. You generate StockOptSpec with the function stockoptspec. This function requires five input arguments. An optional sixth argument InterpMethod, specifying the interpolation method, can be included. The syntax for calling stockoptspec is:

```
[StockOptSpec] = stockoptspec(OptPrice, Strike, Settle, Maturity, OptSpec)
```

where:

- Optprice is a NINST-by-1 vector of European option prices.
- Strike is a NINST-by-1 vector of strike prices.
- Settle is a scalar date marking the settlement date.
- Maturity is a NINST-by-1 vector of maturity dates.
- OptSpec is a NINST-by-1 cell array of strings 'call' or 'put'.

Option Stock Structure Example Using an Implied Trinomial Tree

Consider the following data quoted from liquid options in the market with varying strikes and maturity. You specify these parameters in MATLAB as:

```
Settle = '01/01/06';  
  
Maturity = ['07/01/06';  
           '07/01/06';  
           '07/01/06';  
           '07/01/06';  
           '01/01/07';  
           '01/01/07';  
           '01/01/07';  
           '01/01/07'];
```



```
'07/01/07';
'07/01/07';
'07/01/07';
'07/01/07';
'01/01/08';
'01/01/08';
'01/01/08';
'01/01/08'];

Strike = [113;
101;
100;
88;
128;
112;
100;
78;
144;
112;
100;
69;
162;
112;
100;
61];

OptPrice = [0;
4.807905472659144;
1.306321897011867;
0.048039195057173;
0;
2.310953054191461;
1.421950392866235;
0.020414826276740;
0;
5.091986935627730;
1.346534812295291;
0.005101325584140;
0;
8.047628153217246;
```

```
1.219653432150932;
0.001041436654748];

OptSpec = { 'call';
            'call';
            'put';
            'put';
            'call';
            'call';
            'put';
            'put';
            'call';
            'call';
            'put';
            'put';
            'call';
            'call';
            'put';
            'put'};

StockOptSpec = stockoptspec(OptPrice, Strike, Settle, Maturity, OptSpec)

StockOptSpec =

    FinObj: 'StockOptSpec'
    OptPrice: [16x1 double]
    Strike: [16x1 double]
    Settle: 732678
    Maturity: [16x1 double]
    OptSpec: {16x1 cell}
    InterpMethod: 'price'
```

Note The algorithm for building the ITT tree requires specifying option prices for all tree nodes. The maturities of those options correspond to those of the tree levels, and the strike to the prices on the tree nodes. The types of option are Calls for the nodes above the central nodes, and Puts for those below and including the central nodes.

Clearly, all these options will not be available in the market, hence making interpolation and extrapolation necessary to obtain the node option prices. The degree to which the tree reflects the market will unavoidably be tied to the results of these interpolations and extrapolations. Keeping in mind that extrapolation is less accurate than interpolation, and more so the further away the extrapolated points are from the data points, the function `itttree` issues a warning with a list of the options for which extrapolation was necessary.

In some cases, it may be desirable to view a list of ideal option prices to form an idea of the ranges needed. This can be achieved by calling the function `itttree` specifying only the first three input arguments. The second output argument is a structure array containing the list of ideal options needed.

Creating an Implied Trinomial Tree

You can now use the `StockSpec`, `TimeSpec`, and `StockOptSpec` structures described in “Stock Structure Example Using an Implied Trinomial Tree” on page 3-10, “TimeSpec Example Using an Implied Trinomial Tree” on page 3-11, and “Option Stock Structure Example Using an Implied Trinomial Tree” on page 3-12 to build an implied trinomial tree (ITT). First, you must define the interest rate term structure. For this example, assume that the interest rate is fixed at 8% annually between the valuation date of the tree (January 1, 2006) until its maturity.

```
Rate = 0.08;
ValuationDate = '01-01-2006';
EndDate = '01-01-2008';

RateSpec = intenvset('StartDates', ValuationDate, 'EndDates', EndDate, ...
    'ValuationDate', ValuationDate, 'Rates', Rate, 'Compounding', -1);
```

To build an ITTree, enter:

```
ITTree = ittree(StockSpec, RateSpec, TimeSpec, StockOptSpec)

ITTree =

    FinObj: 'ITStockTree'
    StockSpec: [1x1 struct]
    StockOptSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
    tObs: [0 0.500000000000000 1 1.500000000000000 2]
    dObs: [732678 732860 733043 733225 733408]
    STree: {1x5 cell}
    Probs: {[3x1 double] [3x3 double] [3x5 double] [3x7 double]}
```

Examining Equity Trees

Financial Derivatives Toolbox uses equity binary and implied trinomial trees to represent prices of equity options and of underlying stocks. At the highest level, these trees have structures wrapped around them. The structures encapsulate information needed to interpret information in the tree.

To examine an equity binary or trinomial tree, load the data in the MAT-file `deriv.mat` into the MATLAB workspace.

```
load deriv.mat
```

Display the list of variables loaded from the MAT-file with the `whos` command.

Name	Size	Bytes	Class	Attributes
BDTInstSet	1x1	15956	struct	
BDTTree	1x1	5138	struct	
BKInstSet	1x1	15946	struct	
BKTree	1x1	5904	struct	
CRRInstSet	1x1	12434	struct	
CRRTree	1x1	5058	struct	
EQPInstSet	1x1	12434	struct	
EQPTree	1x1	5058	struct	
HJMInstSet	1x1	15948	struct	

HJMTree	1x1	5838	struct
HWInstSet	1x1	15946	struct
HWTree	1x1	5904	struct
ITTIInstSet	1x1	12438	struct
ITTTTree	1x1	8862	struct
ZeroInstSet	1x1	10282	struct
ZeroRateSpec	1x1	1580	struct

Examining a CRRTree

You can examine in some detail the contents of the CRRTree structure contained in this file.

```
CRRTree

    FinObj: 'BinStockTree'
    Method: 'CRR'
StockSpec: [1x1 struct]
TimeSpec: [1x1 struct]
RateSpec: [1x1 struct]
    tObs: [0 1 2 3 4]
    dObs: [731582 731947 732313 732678 733043]
    STree: {[100] [110.5171 90.4837] [122.1403 100 81.8731]
            [1x4 double] [1x5 double]}
    UpProbs: [0.7309 0.7309 0.7309 0.7309]
```

The Method field of the structure indicates that this is a CRR tree, not an EQP tree.

The fields StockSpec, TimeSpec and RateSpec hold the original structures passed into the function `crrtree`. They contain all the context information required to interpret the tree data.

The fields `tObs` and `dObs` are vectors containing the observation times and dates, the times and dates of the levels of the tree. In this particular case, `tObs` reveals that the tree has a maturity of 4 years (`tObs(end) = 4`) and that it has four time steps (the length of `tObs` is five).

The field `dObs` shows the specific dates for the tree levels, with a granularity of 1 day. This means that all values in `tObs` that correspond to a given day

from 00:00 hours to 24:00 hours are mapped to the corresponding value in `dObs`. You can use the function `datestr` to convert these MATLAB serial dates into their string representations.

The field `UpProbs` is a vector representing the probabilities for up movements from any node in each level. This vector has one element per tree level. All nodes for a given level have the same probability of an up movement. In the specific case being examined, the probability of an up movement is 0.7309 for all levels, and the probability for a down movement is 0.2691 (1 - 0.7309).

Finally, the field `STree` contains the actual stock tree. It is represented in MATLAB as a cell array with each cell array element containing a vector of prices corresponding to a tree level. The prices are in descending order, that is, `CRRTree.STree{3}(1)` represents the topmost element of the third level of the tree, and `CRRTree.STree{3}(end)` represents the bottom element of the same level of the tree.

Examining an ITTree

You can examine in some detail the contents of the `ITTree` structure contained in this file.

```
ITTree =  
  
    FinObj: 'ITStockTree'  
    StockSpec: [1x1 struct]  
    StockOptSpec: [1x1 struct]  
    TimeSpec: [1x1 struct]  
    RateSpec: [1x1 struct]  
    tObs: [0 1 2 3 4]  
    dObs: [732678 733043 733408 733773 734139]  
    STree: {1x5 cell}  
    Probs: {[3x1 double] [3x3 double] [3x5 double] [3x7 double]}
```

The fields `StockSpec`, `StockOptSpec`, `TimeSpec`, and `RateSpec` hold the original structures passed into the function `ittree`. They contain all the context information required to interpret the tree data.

The fields `tObs` and `dObs` are vectors containing the observation times and dates, the times and dates of the levels of the tree. In this particular case,

`tObs` reveals that the tree has a maturity of 4 years (`tObs(end) = 4`) and that it has four time steps (the length of `tObs` is five).

The field `dObs` shows the specific dates for the tree levels, with a granularity of 1 day. This means that all values in `tObs` that correspond to a given day from 00:00 hours to 24:00 hours are mapped to the corresponding value in `dObs`. You can use the function `datestr` to convert these MATLAB serial dates into their string representations.

The field `Probs` is a vector representing the probabilities for movements from any node in each level. This vector has three elements per tree node. In the specific case being examined, at `tObs= 1`, the probability for an up movement is 0.4675, and the probability for a down movement is 0.1934.

Finally, the field `STree` contains the actual stock tree. It is represented in MATLAB as a cell array with each cell array element containing a vector of prices corresponding to a tree level. The prices are in descending order, that is, `ITTree.STree{4}(1)` represents the top element of the fourth level of the tree, and `ITTree.STree{4}(end)` represents the bottom element of the same level of the tree.

Isolating a Specific Node for a CRRTree

The function `treepath` can isolate a specific set of nodes of a binary tree by specifying the path used to reach the final node. As an example, consider the nodes touched by starting from the root node, then following a down movement, then an up movement, and finally a down movement. You use a vector to specify the path, with 1 corresponding to an up movement and 2 corresponding to a down movement. An up-down-up path is then represented as `[2 1 2]`. To obtain the values of all nodes touched by this path, enter:

```
SVals = treepath(CRRTree.STree, [2 1 2])
```

```
SVals =
```

```
100.0000
 90.4837
100.0000
 90.4837
```

The first value in the vector `SVals` corresponds to the root node, and the last value corresponds to the final node reached by following the path indicated.

Isolating a Specific Node for an ITTree

The function `trintreepath` can isolate a specific set of nodes of a trinomial tree by specifying the path used to reach the final node. As an example, consider the nodes touched by starting from the root node, then following an up movement, then a middle movement, and finally a down movement. You use a vector to specify the path, with 1 corresponding to an up movement, 2 corresponding to a middle movement, and 3 corresponding to a down movement. An up-down-middle-down path is then represented as [1 3 2 3]. To obtain the values of all nodes touched by this path, enter:

```
pathSVals = trintreepath(ITTree, [1 3 2 3])

pathSVals =

    50.0000
    66.3448
    50.0000
    50.0000
    37.6819
```

The first value in the vector `pathSVals` corresponds to the root node, and the last value corresponds to the final node reached by following the path indicated.

Differences Between CRR and EQP Tree Structures

In essence, the structures representing CRR trees and EQP trees are similar. If you create a CRR or an EQP tree using identical input arguments, only a few of the tree structure fields differ:

- The `Method` field has a value of 'CRR' or 'EQP' indicating the method used to build the structure.
- The prices in the `STree` cell array have the same structure, but the prices within the cell array are different.

- For EQP, the structure field UpProb always holds a vector with all elements set to 0.5, while for CRR, these probabilities are calculated based on the input arguments passed when building the tree.

Understanding Equity Exotic Options

Financial Derivatives Toolbox supports five types of equity exotic options:

- “Asian Option” on page 3-22
- “Barrier Option” on page 3-23
- “Compound Option” on page 3-24
- “Lookback Option” on page 3-24
- “Bermuda Put and Call Schedule” on page 3-25

Support for all of these equity exotic option types additionally includes American and European puts and calls.

Asian Option

An Asian option is a path-dependent option with a payoff linked to the average value of the underlying asset during the life (or some part of the life) of the option. They are similar to lookback options in that there are two types of Asian options: fixed (average price option) and floating (average strike option). Fixed Asian options have a specified strike, while floating Asian options have a strike equal to the average value of the underlying asset over the life of the option.

There are four Asian options types, each with its own characteristic payoff formula:

- Fixed call: $\max(0, S_{av} - X)$
- Fixed put: $\max(0, X - S_{av})$
- Floating call: $\max(0, S - S_{av})$
- Floating put: $\max(0, S_{av} - S)$

where:

S_{av} is the average price of underlying stock found along the particular path followed to the node.

S is the price of the underlying stock on the node.

X is the strike price (applicable only to fixed Asian options).

S_{av} is defined using either a geometric or an arithmetic average.

Barrier Option

A barrier option is similar to a vanilla put or call option, but its life either begins or ends when the price of the underlying stock passes a predetermined barrier value. There are four types of barrier options.

Up Knock-In

This option becomes effective when the price of the underlying stock passes above a barrier that is above the initial stock price. Once the barrier has knocked in, it will not knock out even if the price of the underlying instrument moves below the barrier again.

Up Knock-Out

This option terminates when the price of the underlying stock passes above a barrier that is above the initial stock price. Once the barrier has knocked out, it will not knock in even if the price of the underlying instrument moves below the barrier again.

Down Knock-In

This option becomes effective when the price of the underlying stock passes below a barrier that is below the initial stock price. Once the barrier has knocked in, it will not knock out even if the price of the underlying instrument moves above the barrier again.

Down Knock-Out

This option terminates when the price of the underlying stock passes below a barrier that is below the initial stock price. Once the barrier has knocked out, it will not knock in even if the price of the underlying instrument moves above the barrier again.

Rebates

If a barrier option fails to exercise, the seller may pay a rebate to the buyer of the option. Knock-outs may pay a rebate when they are knocked out, and knock-ins may pay a rebate if they expire without ever knocking in.

Compound Option

A compound option is basically an option on an option; it gives the holder the right to buy or sell another option. With a compound option, a vanilla stock option serves as the underlying instrument. Compound options thus have two strike prices and two exercise dates.

There are four types of compound options:

- Call on a call
- Put on a put
- Call on a put
- Put on a call

Note The payoff formulas for compound options are too complex for this discussion. If you are interested in the details, consult the paper by Mark Rubinstein entitled “Double Trouble,” published in *Risk* 5 (1991).

Consider the third type, a call on a put. It gives the holder the right to buy a put option. In this case, on the first exercise date, the holder of the compound option is allowed to pay the first strike price and receive a put option. The put option gives the holder the right to sell the underlying asset for the second strike price on the second exercise date.

Lookback Option

A lookback option is a path-dependent option based on the maximum or minimum value the underlying asset achieves during the entire life of the option.

Financial Derivatives Toolbox supports two types of lookback options: fixed and floating. Fixed lookback options have a specified strike price, while floating lookback options have a strike price determined by the asset path. Consequently, there are a total of four lookback option types, each with its own characteristic payoff formula:

- Fixed call: $\max(0, S_{max} - X)$
- Fixed put: $\max(0, X - S_{min})$
- Floating call: $\max(0, S - S_{min})$
- Floating put: $\max(0, S_{max} - S)$

where:

S_{max} is the maximum price of underlying stock found along the particular path followed to the node.

S_{min} is the minimum price of underlying stock found along the particular path followed to the node.

S is the price of the underlying stock on the node.

X is the strike price (applicable only to fixed lookback options).

Bermuda Put and Call Schedule

A Bermuda option is somewhat like a hybrid of American and European options. It can be exercised on predetermined dates only, usually once a month. In Financial Derivatives Toolbox, the relevant information for a Bermuda option is indicated in two input matrices:

- `Strike` — Contains the strike price values for the option.
- `ExerciseDates` — Contains the schedule when the option can be exercised.

Computing Prices and Sensitivities for Equity Derivatives

This section explains how to use Financial Derivatives Toolbox to compute prices and sensitivities of vanilla options and several types of equity exotic options, based on binary and trinomial trees.

- “Computing Instrument Prices” on page 3-26
- “Computing Prices Using CRR” on page 3-28
- “Computing Prices Using EQP” on page 3-30
- “Computing Prices Using ITT” on page 3-32
- “Examining Output from the Pricing Functions” on page 3-34
- “Computing Instrument Sensitivities” on page 3-38
- “Graphical Representation of CRR, EQP, and ITT Trees” on page 3-42

Computing Instrument Prices

The portfolio pricing functions `crrprice`, `eqpprice`, and `ittprice` calculate the price of any set of supported instruments based on a binary equity price tree or an implied trinomial price tree. These functions are capable of pricing the following instrument types:

- Vanilla stock options
 - American and European puts and calls
- Exotic options
 - Asian
 - Barrier
 - Compound
 - Lookback
 - Stock options (Bermuda put and call schedules)

The syntax for calling the function `crrprice` is:

```
[Price, PriceTree] = crrprice(CRRTree, InstSet, Options)
```

The syntax for `eqpprice` is:

```
[Price, PriceTree] = eqpprice(EQPTree, InstSet, Options)
```

The syntax for `ittprice` is:

```
Price = ittprice(ITTTree, ITTInstSet, Options)
```

These functions require two input arguments: the equity price tree and the set of instruments, `InstSet`, and allow a third optional argument.

Required Arguments

`CRRTree` is a CRR equity price tree created using `crrtree`. `EQPTree` is an equal probability equity price tree created using `eqptree`. `ITTTree` is an ITT equity price tree created using `itttree`. See “Building Equity Binary Trees” on page 3-3 and “Building Implied Trinomial Trees” on page 3-8 to learn how to create these structures.

`InstSet` is a structure that represents the set of instruments to be priced independently using the model. Chapter 1, “Getting Started” explains how to create this variable.

Optional Argument

You can enter a third optional argument, `Options`, used when pricing barrier options. For more specific information, see Appendix A, “Derivatives Pricing Options”.

These pricing functions internally classify the instruments and call the appropriate individual instrument pricing function for each of the instrument types. The CRR pricing functions are `asianbycrr`, `barrierbycrr`, `compoundbycrr`, `lookbackbycrr`, and `optstockbycrr`. A similar set of functions exists for EQP and ITT pricing. You can also use these functions directly to calculate the price of sets of instruments of the same type. See the reference pages for these individual functions for further information.

Computing Prices Using CRR

Consider the following example, which uses the portfolio and stock price 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.

Name	Size	Bytes	Class	Attributes
BDTInstSet	1x1	15956	struct	
BDTTree	1x1	5138	struct	
BKInstSet	1x1	15946	struct	
BKTree	1x1	5904	struct	
CRRInstSet	1x1	12434	struct	
CRRTree	1x1	5058	struct	
EQPInstSet	1x1	12434	struct	
EQPTree	1x1	5058	struct	
HJMInstSet	1x1	15948	struct	
HJMTree	1x1	5838	struct	
HWInstSet	1x1	15946	struct	
HWTTree	1x1	5904	struct	
ITTInstSet	1x1	12438	struct	
ITTree	1x1	8862	struct	
ZeroInstSet	1x1	10282	struct	
ZeroRateSpec	1x1	1580	struct	

`CRRTree` and `CRRInstSet` are the required input arguments to call the function `crrprice`.

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

```
instdisp(CRRInstSet)
```


Index	Type	OptSpec	Strike	Settle	ExerciseDates	AmericanOpt	Name	Quantity			
1	OptStock	call	105	01-Jan-2003	01-Jan-2005	1	Call1	10			
2	OptStock	put	105	01-Jan-2003	01-Jan-2006	0	Put1	5			
Index	Type	OptSpec	Strike	Settle	ExerciseDates	AmericanOpt	BarrierSpec	Barrier	Rebate	Name	Quantity
3	Barrier	call	105	01-Jan-2003	01-Jan-2006	1	ui	102	0	Barrier1	1
Index	Type	UOptSpecCOptSpec	CStrike	CSettle	CExerciseDates	CAmericanOpt	Name	Quantity		
4	Compound	callput	5	01-Jan-2003	01-Jan-2005	1	Compound1	3		
Index	Type	OptSpec	Strike	Settle	ExerciseDates	AmericanOpt	Name	Quantity			
5	Lookback	call	115	01-Jan-2003	01-Jan-2006	0	Lookback1	7			
6	Lookback	call	115	01-Jan-2003	01-Jan-2007	0	Lookback2	9			
Index	Type	OptSpec	Strike	Settle	ExerciseDates	AmericanOpt	AvgType	AvgPrice	AvgDate	Name	Quantity
7	Asian	put	110	01-Jan-2003	01-Jan-2006	0	arithmetic	NaN	NaN	Asian1	4
8	Asian	put	110	01-Jan-2003	01-Jan-2007	0	arithmetic	NaN	NaN	Asian2	6

Note Because of space considerations, the compound option above (Index 4) has been condensed to fit the page. The `instdispinstdisp` command displays all compound option fields on your computer screen.

The instrument set contains eight instruments:

- Two vanilla options (Call1, Put1)
- One barrier option (Barrier1)
- One compound option (Compound1)
- Two lookback options (Lookback1, Lookback2)
- Two Asian options (Asian1, Asian2)

Each instrument has a corresponding index that identifies the instrument prices in the price vector returned by `crrprice`.

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

```
Price = crrprice(CRRTree, CRRInstSet)
```

```
Price =
```

```
8.2863
2.5016
12.1272
3.3241
7.6015
11.7772
4.1797
3.4219
```

Computing Prices Using EQP

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.

Name	Size	Bytes	Class	Attributes
BDTInstSet	1x1	15956	struct	
BDTTree	1x1	5138	struct	
BKInstSet	1x1	15946	struct	
BKTree	1x1	5904	struct	
CRRInstSet	1x1	12434	struct	
CRRTree	1x1	5058	struct	
EQPInstSet	1x1	12434	struct	
EQPTree	1x1	5058	struct	
HJMInstSet	1x1	15948	struct	
HJMTree	1x1	5838	struct	
HWInstSet	1x1	15946	struct	
HWTree	1x1	5904	struct	
ITTIInstSet	1x1	12438	struct	
ITTITree	1x1	8862	struct	
ZeroInstSet	1x1	10282	struct	
ZeroRateSpec	1x1	1580	struct	

EQPTree and EQPInstSet are the input arguments required to call the function `eqpprice`.

Use the command `instdisp` to examine the set of instruments contained in the variable `EQPInstSet`.

```
instdisp(EQPInstSet)
```

```

Index Type      OptSpec Strike Settle      ExerciseDates AmericanOpt Name  Quantity
1      OptStock call   105   01-Jan-2003  01-Jan-2005  1      Call1 10
2      OptStock put    105   01-Jan-2003  01-Jan-2006  0      Put1  5

Index Type      OptSpec Strike Settle      ExerciseDates AmericanOpt BarrierSpec Barrier Rebate Name  Quantity
3      Barrier call   105   01-Jan-2003  01-Jan-2006  1      ui     102  0  Barrier1 1

Index Type      UOptSpec ...COptSpec CStrike CSettle      CExerciseDates CAmericanOpt Name  Quantity
4      Compound call   ...put  5      01-Jan-2003  01-Jan-2005  1      Compound1 3

Index Type      OptSpec Strike Settle      ExerciseDates AmericanOpt Name  Quantity
5      Lookback call   115   01-Jan-2003  01-Jan-2006  0      Lookback1 7
6      Lookback call   115   01-Jan-2003  01-Jan-2007  0      Lookback2 9

Index Type      OptSpec Strike Settle      ExerciseDates AmericanOpt AvgType  AvgPrice AvgDate Name  Quantity
7      Asian put      110   01-Jan-2003  01-Jan-2006  0      arithmetic NaN   NaN   Asian1 4
8      Asian put      110   01-Jan-2003  01-Jan-2007  0      arithmetic NaN   NaN   Asian2 6

```

Note Because of space considerations, the compound option above (Index 4) has been condensed to fit the page. The `instdisp` command displays all compound option fields on your computer screen.

The instrument set contains eight instruments:

- Two vanilla options (Call1, Put1)
- One barrier option (Barrier1)
- One compound option (Compound1)
- Two lookback options (Lookback1, Lookback2)
- Two Asian options (Asian1, Asian2)

Each instrument has a corresponding index that identifies the instrument prices in the price vector returned by `eqpprice`.

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

```
Price = eqpprice(EQPtree, EQPInstSet)
```

```
Price =
```

```
    8.4791  
    2.6375  
   12.2632  
    3.5091  
    8.7941  
   12.9577  
    4.7444  
    3.9178
```

Computing Prices Using ITT

Consider the following example, which uses the portfolio and stock price 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.

Name	Size	Bytes	Class	Attributes
BDTInstSet	1x1	15956	struct	
BDTTree	1x1	5138	struct	
BKInstSet	1x1	15946	struct	
BKTree	1x1	5904	struct	
CRRInstSet	1x1	12434	struct	
CRRTree	1x1	5058	struct	
EQPInstSet	1x1	12434	struct	
EQPtree	1x1	5058	struct	
HJMInstSet	1x1	15948	struct	
HJMTree	1x1	5838	struct	
HWInstSet	1x1	15946	struct	
HWTree	1x1	5904	struct	
ITTInstSet	1x1	12438	struct	
ITTTree	1x1	8812	struct	
ZeroInstSet	1x1	10282	struct	

```
ZeroRateSpec      1x1      1580 struct
```

ITTree and ITInstSet are the input arguments required to call the function `ittprice`. Use the command `instdisp` to examine the set of instruments contained in the variable `ITInstSet`.

```
instdisp(ITInstSet)
```

```

Index Type      OptSpec Strike Settle      ExerciseDates AmericanOpt Name      Quantity
1  OptStock call   95   01-Jan-2006  31-Dec-2008  1      Call1 10
2  OptStock put    80   01-Jan-2006  01-Jan-2010  0      Put1  4

Index Type      OptSpec Strike Settle      ExerciseDates AmericanOpt BarrierSpec Barrier Rebate Name      Quantity
3  Barrier call   85   01-Jan-2006  31-Dec-2008  1      ui      115   0      Barrier1 1

Index Type      UOptSpec UStrike USettle      UExerciseDates UAmericanOpt COptSpec CStrike CSettle      CExerciseDates CAmericanOpt Name      Quantity
4  Compound call   99   01-Jan-2006  01-Jan-2010  1      put    5      01-Jan-2006  01-Jan-2010  1      Compound1 3

Index Type      OptSpec Strike Settle      ExerciseDates AmericanOpt Name      Quantity
5  Lookback call   85   01-Jan-2006  01-Jan-2008  0      Lookback1 7
6  Lookback call   85   01-Jan-2006  01-Jan-2010  0      Lookback2 9

Index Type      OptSpec Strike Settle      ExerciseDates AmericanOpt AvgType      AvgPrice AvgDate Name      Quantity
7  Asian call    55   01-Jan-2006  01-Jan-2008  0      arithmetic NaN      NaN      Asian1 5
8  Asian call    55   01-Jan-2006  01-Jan-2010  0      arithmetic NaN      NaN      Asian2 7

```

The instrument set contains eight instruments:

- Two vanilla options (Call1, Put1)
- One barrier option (Barrier1)
- One compound option (Compound1)
- Two lookback options (Lookback1, Lookback2)
- Two Asian options (Asian1, Asian2)

Each instrument has a corresponding index that identifies the instrument prices in the price vector returned by `ittprice`.

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

```
Price = ittprice(ITTree, ITInstSet)
```

```
Price =
```

```
1.650
10.68
2.407
3.229
0.542
6.184
3.205
6.607
```

Examining Output from the Pricing Functions

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 equity tree. The instrument indexing within `Price` is the same as the indexing within `InstSet`.

In the CRR example, the prices in the `Price` vector correspond to the instruments in this order.

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

InstNames =

Call1
Put1
Barrier1
Compound1
Lookback1
Lookback2
Asian1
Asian2
```

Consequently, in the `Price` vector, the fourth element, 3.3241, represents the price of the fourth instrument (`Compound1`), and the sixth element, 11.7772, represents the price of the sixth instrument (`Lookback2`).

In the ITT example, the prices in the `Price` vector correspond to the instruments in this order.

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

```

InstNames =

Call1
Put1
Barrier1
Compound1
Lookback1
Lookback2
Asian1
Asian2

```

Consequently, in the `Price` vector, the first element, 1.650, represents the price of the first instrument (`Call1`), and the eighth element, 6.607, represents the price of the eighth instrument (`Asian2`).

Price Tree Output for CRR

If you call a pricing function with two output arguments, for example:

```
[Price, PriceTree] = crrprice(CRRTree, CRRInstSet)
```

you generate a price tree structure along with the price information.

This price tree structure `PriceTree` holds all pricing information.

```

PriceTree =
FinObj: 'BinPriceTree'
PTree: {[8x1 double] [8x2 double] [8x3 double] [8x4 double] [8x5 double]}
tObs: [0 1 2 3 4]
dObs: [731582 731947 732313 732678 733043]

```

The first field of this structure, `FinObj`, indicates that this structure represents a price tree. The second field, `PTree`, is the tree holding the prices of the instruments in each node of the tree. Finally, the third and fourth fields, `tObs` and `dObs`, represent the observation time and date of each level of `PTree`, with `tObs` using units in terms of compounding periods.

Using the command-line interface, you can directly examine `PriceTree.PTree`, 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.PTree{1}
ans =
 8.2863
 2.5016
12.1272
 3.3241
 7.6015
11.7772
 4.1797
 3.4219
```

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

The function `eqpprice` also returns a price tree that you can examine in the same way.

Price Tree Output for ITT

If you call a pricing function with two output arguments, for example:

```
[Price, PriceTree] = ittprice(ITTTree, ITTInstSet)
```

you generate a price tree structure along with the price information.

This price tree structure `PriceTree` holds all pricing information.

```
PriceTree =

  FinObj: 'TrinPriceTree'
  PTree: {[8x1 double] [8x3 double] [8x5 double] [8x7 double] [8x9 double]}
  tObs: [0 1 2 3 4]
  dObs: [732678 733043 733408 733773 734139]
```

The first field of this structure, `FinObj`, indicates that this structure represents a trinomial price tree. The second field, `PTree` is the tree holding the prices of the instruments in each node of the tree. Finally, the third and fourth fields, `tObs` and `dObs`, represent the observation time and date of each level of `PTree`, with `tObs` using units in terms of compounding periods.

Using the command-line interface, you can directly examine `PriceTree.PTree`, 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.PTree{1}

    1.6506
   10.6832
    2.4074
    3.2294
    0.5426
    6.1845
    3.2052
    6.6074
```

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

Prices for Lookback and Asian Options for Equity Trees

Lookback options and Asian options are path dependent, and, as such, there are no unique prices for any node except the root node. Consequently, the corresponding values for lookback and Asian options in the price tree are set to `NaN`, the only exception being the root node. This becomes apparent if you examine the prices in the second node (`tObs = 1`) of the CRR price tree:

```
PriceTree.PTree{2}

ans =

    11.9176     0
     0.9508     7.1914
    16.4600     2.6672
     2.5896     5.0000
         NaN     NaN
         NaN     NaN
         NaN     NaN
         NaN     NaN
```

Examining the prices in the second node (`tobs = 1`) of the ITT price tree displays:

```
PriceTree.PTree{2}

ans =

    3.9022         0         0
    6.3736    13.3743    22.1915
    5.6914         0         0
    2.7663     3.8594     5.0000
    NaN         NaN         NaN
    NaN         NaN         NaN
    NaN         NaN         NaN
    NaN         NaN         NaN
```

Computing Instrument Sensitivities

Sensitivities can be reported either as dollar price changes or percentage price changes. The delta, gamma, and vega sensitivities that the toolbox computes are dollar sensitivities.

The functions `crrsens`, `eqpsens`, and `ittsens` compute the delta, gamma, and vega sensitivities of instruments using a stock tree. They also optionally return the calculated price for each instrument. The sensitivity functions require the same two input arguments used by the pricing functions (`CRRTree` and `CRRInstSet` for CRR, `EQPTree` and `EQPInstSet` for EQP, and `ITTTree` and `ITTInstSet` for ITT).

As with the instrument pricing functions, the optional input argument `Options` is also allowed. You would include this argument if you want a sensitivity function to generate a price for a barrier option as one of its outputs and want to control the method that the toolbox uses to perform the pricing operation. See Appendix A, “Derivatives Pricing Options” or the `derivset` function for more information.

For path-dependent options (lookback and Asian), delta and gamma are computed by finite differences in calls to `crrprice`, `eqpprice`, and `ittprice`. For the other options (stock option, barrier, and compound), delta and gamma are computed from the CRR, EQP, and ITT trees and the corresponding option price tree. (See Chriss, Neil, *Black-Scholes and Beyond*, pp. 308-312.)

CRR Sensitivities Example

The calling syntax for the sensitivity function is:

```
[Delta, Gamma, Vega, Price] = crrsens(CRRTree, InstSet, Options)
```

Using the example data in `deriv.mat`, calculate the sensitivity of the instruments.

```
load deriv.mat
[Delta, Gamma, Vega, Price] = crrsens(CRRTree, CRRInstSet);
```

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

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

All =

    0.59         0.04    53.45         8.29
   -0.31         0.03    67.00         2.50
    0.69         0.03    67.00        12.13
   -0.12        -0.01   -98.08         3.32
   -0.40   -45926.32    88.18         7.60
   -0.42  -112143.15   119.19        11.78
    0.60    45926.32    49.21         4.18
    0.82   112143.15    41.71         3.42
```

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

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

    0.07         0.00     6.45         8.29
   -0.12         0.01    26.78         2.50
    0.06         0.00     5.53        12.13
   -0.04        -0.00   -29.51         3.32
```

-0.05	-6041.77	11.60	7.60
-0.04	-9522.02	10.12	11.78
0.14	10987.98	11.77	4.18
0.24	32771.92	12.19	3.42

ITT Sensitivities Example

The calling syntax for the sensitivity function is:

```
[Delta, Gamma, Vega, Price] = ittens(ITTtree, ITTInstSet,
Options)
```

Using the example data in `deriv.mat`, calculate the sensitivity of the instruments.

```
load deriv.mat
warning('off', 'finderiv:itttree:Extrapolation');
[Delta, Gamma, Vega, Price] = ittens(ITTtree, ITTInstSet);
```

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

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

```
All =
```

0.24	0.03	19.35	1.65
-0.43	0.02	49.69	10.68
0.35	0.04	12.29	2.41
-0.07	0.00	6.73	3.23
0.63	142945.66	38.90	0.54
0.60	22703.21	68.92	6.18
0.32	-142945.66	18.48	3.21
0.67	-22703.21	17.75	6.61

As with the prices, each row of the sensitivity vectors corresponds to the similarly indexed instrument in `ITTInstSet`.

Note In this example, the extrapolation warnings are turned off before calculating the sensitivities to avoid displaying a large number of warnings on the Command Window as the sensitivities are calculated.

If the extrapolation warnings are turned on

```
warning('on', 'finderiv:itttree:Extrapolation');
```

and ittsens is rerun, the extrapolation warnings scroll as the command executes:

```
[Delta, Gamma, Vega, Price] = ittsens(ITTTree, ITTInstSet)
```

```
Warning: The option set specified in StockOptSpec was too narrow for the generated tree.
This makes extrapolation necessary. The list of options outside of the
range of those specified in StockOptSpec are:
```

```
Option Type: 'call'   Maturity: 01-Jan-2007   Strike=66.3529
Option Type: 'put'   Maturity: 01-Jan-2007   Strike=50.0061
Option Type: 'put'   Maturity: 01-Jan-2008   Strike=50.0061
Option Type: 'put'   Maturity: 31-Dec-2008   Strike=50.0061
Option Type: 'call'  Maturity: 01-Jan-2010   Strike=155.0141
Option Type: 'put'   Maturity: 01-Jan-2010   Strike=50.006
> In itttree>InterpOptPrices at 675
  In itttree at 277
  In stocktreesens>stocktreedeltagamma_PD at 127
  In stocktreesens at 83
  In ittsens at 81
```

```
Warning: The option set specified in StockOptSpec was too narrow for the generated tree.
This made extrapolation necessary. Below is a list of the options that were outside of the
range of those specified in StockOptSpec.
```

```
Option Type: 'call'   Maturity: 01-Jan-2007   Strike=66.3367
Option Type: 'put'   Maturity: 01-Jan-2007   Strike=37.6773
Option Type: 'call'  Maturity: 01-Jan-2008   Strike=66.3367
Option Type: 'put'   Maturity: 01-Jan-2008   Strike=28.3951
Option Type: 'call'  Maturity: 31-Dec-2008   Strike=66.3367
Option Type: 'call'  Maturity: 01-Jan-2010   Strike=66.3367
```

```
Option Type: 'put'    Maturity: 01-Jan-2010  Strike=16.1276
```

```
> In itttree>InterpOptPrices at 675
  In itttree at 277
  In stocktreesens>stocktreedeltagamma_PD at 131
  In stocktreesens at 83
  In itttsens at 81
```

Warning: The option set specified in StockOptSpec was too narrow for the generated tree. This made extrapolation necessary. Below is a list of the options that were outside of the range of those specified in StockOptSpec.

```
Option Type: 'call'   Maturity: 01-Jan-2007  Strike=67.2897
Option Type: 'put'    Maturity: 01-Jan-2007  Strike=37.1528
Option Type: 'put'    Maturity: 01-Jan-2008  Strike=27.6066
Option Type: 'put'    Maturity: 31-Dec-2008  Strike=20.5132
Option Type: 'call'   Maturity: 01-Jan-2010  Strike=164.0157
Option Type: 'put'    Maturity: 01-Jan-2010  Strike=15.2424
```

```
> In itttree>InterpOptPrices at 675
  In itttree at 277
  In stocktreesens>stocktreevega at 191
  In stocktreesens at 92
  In itttsens at 81
```

These warnings are a consequence of having to extrapolate to find the option price of the tree nodes. In this example, the set of inputs options was too narrow for the shift in the tree nodes introduced by the disturbance used to calculate the sensitivities. As a consequence extrapolation for some of the nodes was needed. Since the input data is quite close the extrapolated data, the error introduced by extrapolation is fairly low.

Graphical Representation of CRR, EQP, and ITT Trees

You can use the function `treeviewer` to display a graphical representation of a tree, allowing you to examine interactively the prices and rates on the nodes of the tree until maturity. The graphical representations of CRR and EQP trees are equivalent to Black-Derman-Toy (BDT) trees, given that they are all binary recombining trees. The graphical representations of ITT trees are equivalent to Hull-White (HW) trees, given that they are all trinomial

recombining trees. See “Graphical Representation of Trees” on page 2-61 for an overview on the use of treeviewer with CRR trees, EQP trees, and ITT trees and their corresponding option price trees. Follow the instructions for BDT trees.

Hedging Portfolios

Hedging (p. 4-2)

Purpose of hedging.

Hedging Functions (p. 4-2)

Two hedging functions: `hedgeopt` and `hedgeslf`.

Specifying Constraints with `ConSet`
(p. 4-15)

Constraints for `hedgeopt` and `hedgeslf`.

Hedging with Constrained Portfolios
(p. 4-20)

Examples of hedging with constrained portfolios.

Hedging

Hedging is an important consideration in modern finance. 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.

If there were no 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 expensive. Therefore, traders and portfolio managers must carefully assess the cost needed to achieve a particular portfolio sensitivity (for example, maintaining delta, gamma, and vega neutrality). Thus, the hedging problem involves the fundamental tradeoff between portfolio insurance and the cost of such insurance coverage.

Hedging Functions

- “Hedging with hedgeopt” on page 4-4
- “Self-Financing Hedges with hedgeslf” on page 4-11

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 4-4.

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 (that is, 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 with `hedgeslf`” on page 4-11.

The examples in this section consider the *delta*, *gamma*, and *vega* sensitivity measures. In this toolbox, when you work with *interest-rate derivatives*, 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. See `bdtsens` or `hjmsens` for details on the computation of sensitivities for interest-rate derivatives.

For *equity exotic options*, the underlying instrument is the stock price instead of the forward yield curve. Consequently, delta now represents the price sensitivity measure of shifts in the stock price, gamma is the delta sensitivity measure of shifts in the stock price, and vega is the price sensitivity measure of shifts in the volatility of the stock. See `crrsens` or `eqpsens` for details on the computation of sensitivities for equity derivatives.

For examples showing the computation of sensitivities for interest-rate based derivatives, see “Computing Instrument Sensitivities” on page 2-26. Likewise, for examples showing the computation of sensitivities for equity exotic options, see “Computing Instrument Sensitivities” on page 3-38.

Note The delta, gamma, and vega sensitivities that the toolbox calculates are dollar sensitivities.

Hedging with hedgeopt

Note The numerical results in this section are displayed in 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

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.

$$\text{Value0} = \text{Holdings}' * \text{Price}$$

$$\text{Value0} =$$

23674.62

$$\text{Value1} = \text{Quantity} * \text{Price}$$

$$\text{Value1} =$$

23674.62

Partially Hedged Portfolio

Building on the example in “Maintaining Existing Allocations” on page 4-5, 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 example in “Partially Hedged Portfolio” on page 4-6 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 examples in “Fully Hedged Portfolio” on page 4-8 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 that the same instruments are held fixed, and that the cost frontier is evaluated from \$0 to \$50,000 at increments of \$1000.

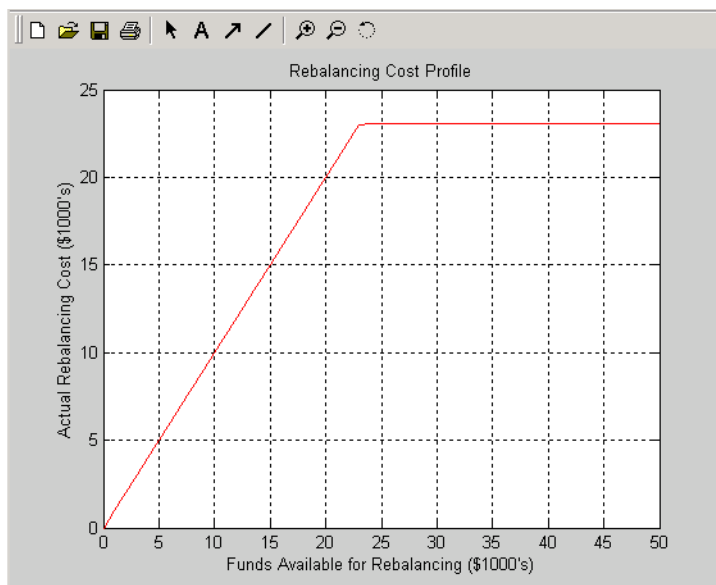
```
MaxCost = [0:1000:50000];
```

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')
```

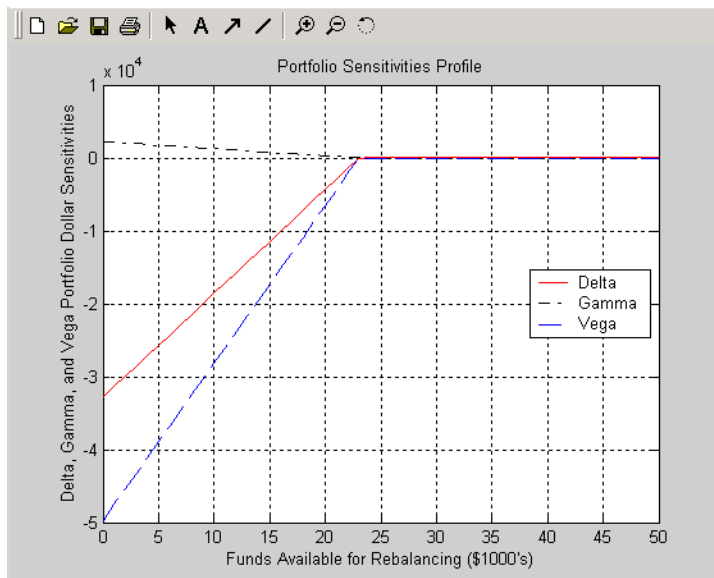


Rebalancing Cost Profile

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)
```



Funds Available for Rebalancing

Self-Financing Hedges with hedgeslf

The figures Rebalancing Cost Profile on page 4-10 and Funds Available for Rebalancing on page 4-11 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 this 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

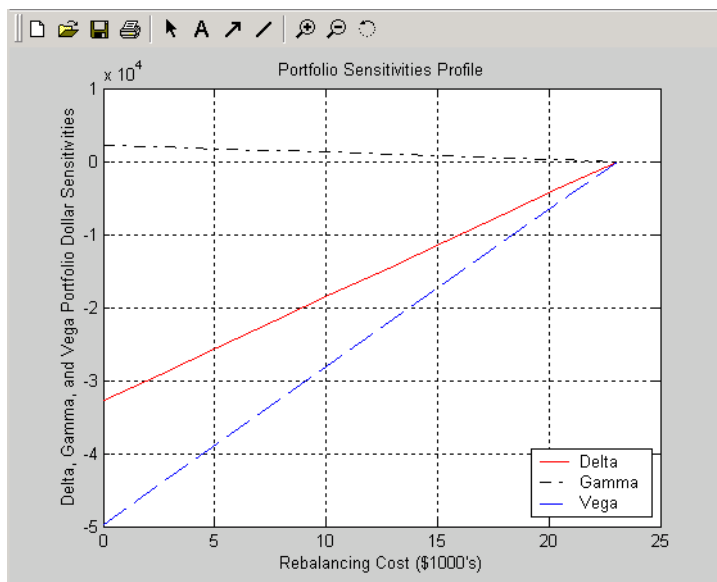
```

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)

```



Rebalancing Cost

In this calling form, `hedgeopt` calls `hedges1f` 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 `hedges1f` cast the optimization problem as a constrained linear least squares problem. Depending on 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 4-20 for additional information.

Specifying Constraints with ConSet

- “Setting Constraints” on page 4-15
- “Portfolio Rebalancing” on page 4-17

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 “Analyzing Portfolios” in the Financial Toolbox documentation.

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 0). 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 next 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 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
```

hedges1f 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 the figure Rebalancing Cost on page 4-14.

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

- “Example: Fully Hedged Portfolio” on page 4-20
- “Example: Minimize Portfolio Sensitivities” on page 4-23
- “Example: Under-Determined System” on page 4-24
- “Example: Portfolio Constraints with hedgeslf” on page 4-25

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

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

where C , A , and Aeq are matrices, and d , b , beq , lb , and ub are vectors. For Financial Derivatives Toolbox, x is a vector of asset holdings (contracts).

Depending on 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, it 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 example in “Fully Hedged Portfolio” on page 4-8 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, for example, 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 (for example, [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 $Z \leq \mathbf{1}$. Furthermore, assume we somehow add another constraint that effectively restricts $Z \leq \mathbf{0}$. The constraint $Z \leq \mathbf{1}$ 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 the figures Rebalancing Cost Profile on page 4-10, Funds Available for Rebalancing on page 4-11, and Rebalancing Cost on page 4-14.

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.

Example: 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.

To illustrate this hedging facility using equity exotic options, consider the portfolio `CRRInstSet` obtained from the example MAT-file `deriv.mat`. The portfolio consists of eight option instruments: two stock options, one barrier, one compound, two lookback, and two Asian.

The hedging functions require inputs that include the current portfolio holdings (allocations) and a matrix of instrument sensitivities. To create these inputs, start by loading the example portfolio into memory

```
load deriv.mat;
```

Next, compute the prices and sensitivities of the instruments in this portfolio.

```
[Delta, Gamma, Vega, Price] = crrsens(CRRTree, CRRInstSet);
```

Extract the current portfolio holdings (the quantity held or the number of contracts).

```
Holdings = instget(CRRInstSet, '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.

```
disp([Price Holdings Sensitivities])
```

```
8.29      10.00      0.59      0.04      53.45
```

2.50	5.00	-0.31	0.03	67.00
12.13	1.00	0.69	0.03	67.00
3.32	3.00	-0.12	-0.01	-98.08
7.60	7.00	-0.40	-45926.32	88.18
11.78	9.00	-0.42	-112143.15	119.19
4.18	4.00	0.60	45926.32	49.21
3.42	6.00	0.82	112143.15	41.71

The first column contains the dollar unit price of each instrument, the second contains the holdings of each instrument, and the third, fourth, and fifth columns contain the delta, gamma, and vega dollar sensitivities, respectively.

Suppose that you want to obtain a delta, gamma and vega neutral portfolio using `hedgeslf`.

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

Sens =

```
0.00
-0.00
0.00
```

Value1 =

```
313.93
```

Quantity =

```
10.00
7.64
-1.56
26.13
9.94
3.73
-0.75
8.11
```

`hedgelf` returns the portfolio dollar sensitivities (`Sens`), the value of the rebalanced portfolio (`Value1`) and the new allocation for each instrument (`Quantity`).

If `Value0` and `Value1` represent the portfolio value before and after rebalancing, respectively, you can verify the cost by comparing the portfolio values.

```
Value0= Holdings' * Price
```

```
Value0 =
```

```
313.93
```

In this example, the portfolio is fully hedged (simultaneous delta, gamma, and vega neutrality) and self-financing (the values of the portfolio before and after balancing (`Value0` and `Value1`) are the same.

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

As an example, assume that, in addition to holding instrument 1 fixed as before, you want to bound the position of all instruments to within +/- 20 contracts (for each instrument, you cannot short or long more than 20 contracts). Applying these constraints disallows the current position in the fourth instrument (long 26.13). 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 = [-20 -20 -20 -20 -20 -20 -20 -20];  
UpperBounds = [20 20 20 20 20 20 20 20];  
ConSet = portcons('AssetLims', LowerBounds, UpperBounds);
```

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

```
[Sens, Cost, Quantity1] = hedgelf(Sensitivities, Price, ...  
Holdings, 1, ConSet)
```

Sens =

-0.00
0.00
0.00

Cost =

313.93

Quantity1 =

10.00
5.28
10.98
20.00
20.00
-6.99
-20.00
9.39

Observe that `hedges1f` enforces the upper bound on the fourth instrument, and the portfolio continues to be fully hedged and self-financing.

Functions — By Category

Portfolio Hedge Allocation (p. 5-2)	Work with hedge portfolios
Price and Sensitivity from Interest-Rate Term Structure (p. 5-2)	Work with interest-rate term structure
Price and Sensitivity from Heath-Jarrow-Morton Trees (p. 5-3)	Work with Heath-Jarrow-Morton trees
Price and Sensitivity from Black-Derman-Toy Trees (p. 5-3)	Work with Black-Derman-Toy trees
Price and Sensitivity from Black-Karasinski Trees (p. 5-4)	Work with Black-Karasinski trees
Price and Sensitivity from Cox-Ross-Rubinstein Trees (p. 5-4)	Work with Cox-Ross-Rubinstein trees
Price and Sensitivity from Equal Probabilities Binomial Trees (p. 5-5)	Working with Equal Probabilities Binomial trees
Price and Sensitivity from Hull-White Trees (p. 5-5)	Price and sensitivity functions for working with Hull-White trees
Price and Sensitivity from Implied Trinomial Tree (p. 5-6)	Price and sensitivity functions for working with Hull-White trees
Heath-Jarrow-Morton Utilities (p. 5-6)	Work with Heath-Jarrow-Morton utilities
Black-Derman-Toy Utilities (p. 5-7)	Work with Black-Derman-Toy utilities
Black-Karasinski Utilities (p. 5-8)	Work with Black-Karasinski utilities

Cox-Ross-Rubinstein Utilities (p. 5-8)	Work with Cox-Ross-Rubinstein utilities
Equal Probabilities Tree Utilities (p. 5-9)	Work with equal probabilities tree utilities
Implied Trinomial Tree Utilities (p. 5-9)	Work with implied trinomial tree utilities
Hull-White Utilities (p. 5-10)	Work with Hull-White utilities
Tree Manipulation (p. 5-10)	General tree manipulation
Derivatives Pricing Options (p. 5-11)	Work with derivatives pricing options
Instrument Portfolio Handling (p. 5-11)	Work with instrument portfolios
Financial Object Structures (p. 5-12)	Work with financial structures
Interest Term Structure (p. 5-13)	Work with interest term structure
Date (p. 5-13)	Display date entries
Graphical Display (p. 5-13)	Display tree information graphically

Portfolio Hedge Allocation

hedgeopt	Allocate optimal hedge for target costs or sensitivities
hedges1f	Self-financing hedge

Price and Sensitivity from Interest-Rate Term Structure

bondbyzero	Price bond from set of zero curves
cfbyzero	Price cash flows from set of zero curves

fixedbyzero	Price fixed-rate note from set of zero curves
floatbyzero	Price floating-rate note from set of zero curves
intenvprice	Price instruments from set of zero curves
intenvsens	Instrument price and sensitivities from set of zero curves
swapbyzero	Price swap instrument from set of zero curves

Price and Sensitivity from Heath-Jarrow-Morton Trees

hjmprice	Instrument prices from HJM interest-rate tree
hjmsens	Instrument prices and sensitivities from HJM interest-rate tree
hjmtimespec	Specify time structure for HJM interest-rate tree
hjmtree	Construct HJM interest-rate tree
hjmvolspec	Specify HJM interest-rate volatility process

Price and Sensitivity from Black-Derman-Toy Trees

bdtpprice	Instrument prices from BDT interest-rate tree
bdtsens	Instrument prices and sensitivities from BDT interest-rate tree

<code>bdttimespec</code>	Specify time structure for BDT interest-rate tree
<code>bdttree</code>	Construct BDT interest-rate tree
<code>bdtvolspec</code>	Specify BDT interest-rate volatility process

Price and Sensitivity from Black-Karasinski Trees

<code>bkprice</code>	Instrument prices from Black-Karasinski interest-rate tree
<code>bksens</code>	Instrument prices and sensitivities from Black-Karasinski interest-rate tree
<code>bktimespec</code>	Specify time structure for Black-Karasinski tree
<code>bktree</code>	Construct Black-Karasinski interest-rate tree
<code>bkvolspec</code>	Specify Black-Karasinski interest-rate volatility process

Price and Sensitivity from Cox-Ross-Rubinstein Trees

<code>crrprice</code>	Instrument prices from CRR tree
<code>crrsens</code>	Instrument prices and sensitivities from CRR tree
<code>crrtimespec</code>	Specify time structure for CRR tree
<code>crrtree</code>	Construct CRR stock tree

Price and Sensitivity from Equal Probabilities Binomial Trees

eqpprice	Instrument prices from EQP binomial tree
eqpsens	Instrument prices and sensitivities from EQP binomial tree
eqptimespec	Specify time structure for EQP binomial tree
eqptree	Construct EQP stock tree

Price and Sensitivity from Hull-White Trees

hwprice	Instrument prices from Hull-White interest-rate tree
hwsens	Instrument prices and sensitivities from HW interest-rate tree
hwtimespec	Specify time structure for Hull-White tree
hwtree	Construct Hull-White interest-rate tree
hwvolspec	Specify Hull-White interest-rate volatility process

Price and Sensitivity from Implied Trinomial Tree

<code>ittprice</code>	Price instruments using an implied trinomial tree (ITT)
<code>ittsens</code>	Instrument sensitivities and prices using an implied trinomial tree (ITT)
<code>itttimespec</code>	Specify time structure using an implied trinomial tree (ITT)
<code>itttree</code>	Build an implied trinomial stock tree
<code>stockoptspec</code>	Specify European stock option structure

Heath-Jarrow-Morton Utilities

<code>bondbyhjm</code>	Price bond from HJM interest-rate tree
<code>capbyhjm</code>	Price cap instrument from HJM interest-rate tree
<code>cfbyhjm</code>	Price cash flows from HJM interest-rate tree
<code>fixedbyhjm</code>	Price fixed-rate note from HJM interest-rate tree
<code>floatbyhjm</code>	Price floating-rate note from HJM interest-rate tree
<code>floorbyhjm</code>	Price floor instrument from HJM interest-rate tree
<code>mmktbyhjm</code>	Create money-market tree from HJM interest-rate tree

optbndbyhjm	Price bond option from HJM interest-rate tree
swapbyhjm	Price swap instrument from HJM interest-rate tree

Black-Derman-Toy Utilities

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

Black-Karasinski Utilities

bondbybk	Price bond from Black-Karasinski interest-rate tree
capbybk	Price cap instrument from Black-Karasinski interest-rate tree
cfbybk	Price cash flows from Black-Karasinski interest-rate tree
fixedbybk	Price fixed-rate note from Black-Karasinski interest-rate tree
floatbybk	Price floating-rate note from Black-Karasinski interest-rate tree
floorbybk	Price floor instrument from Black-Karasinski interest-rate tree
optbndbybk	Price bond option from Black-Karasinski interest-rate tree
swapbybk	Price swap instrument from Black-Karasinski interest-rate tree

Cox-Ross-Rubinstein Utilities

asianbycrr	Price Asian option from CRR binomial tree
barrierbycrr	Price barrier option from CRR binomial tree
compoundbycrr	Price compound option from CRR binomial tree

lookbackbycrr

Price lookback option from CRR tree

optstockbycrr

Price stock option from CRR tree

Equal Probabilities Tree Utilities

asianbyeqp

Price Asian option from EQP
binomial tree

barrierbyeqp

Price barrier option from EQP
binomial tree

compoundbyeqp

Price compound option from EQP
binomial tree

lookbackbyeqp

Price lookback option from EQP
binomial tree

optstockbyeqp

Price stock option from EQP
binomial tree

Implied Trinomial Tree Utilities

asianbyitt

Price Asian options using an implied
trinomial tree (ITT)

barrierbyitt

Price barrier options using an
implied trinomial tree (ITT)

compoundbyitt

Price compound options using an
implied trinomial tree (ITT)

lookbackbyitt

Price lookback option using an
implied trinomial tree (ITT)

optstockbyitt

Price options on stocks using an
implied trinomial tree (ITT)

Hull-White Utilities

bondbyhw	Price bond from Hull-White interest-rate tree
capbyhw	Price cap instrument from Hull-White interest-rate tree
cfbyhw	Price cash flows from Hull-White interest-rate tree
fixedbyhw	Price fixed-rate note from Hull-White interest-rate tree
floatbyhw	Price floating-rate note from Hull-White interest-rate tree
floorbyhw	Price floor instrument from Hull-White interest-rate tree
optbndbyhw	Price bond option from Hull-White interest-rate tree
swapbyhw	Price swap instrument from Hull-White interest-rate tree

Tree Manipulation

bushpath	Extract entries from node of bushy tree
bushshape	Retrieve shape of bushy tree
cvtree	Convert inverse-discount tree to interest-rate tree
mkbush	Create bushy tree
mktree	Create recombining binomial tree
mktrintree	Create recombining trinomial tree

treepath	Entries from node of recombining binomial tree
treeshape	Shape of recombining binomial tree
trintreepath	Entries from node of recombining trinomial tree
trintreeshape	Shape of recombining trinomial 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 instrument collection
instasian	Construct Asian option
instbarrier	Construct barrier option
instbond	Construct bond instrument
instcap	Construct cap instrument
instcf	Construct cash flow instrument
instcompound	Construct compound option
instdelete	Complement of instrument set by matching conditions
instdisp	Display instruments

<code>instfields</code>	List field names
<code>instfind</code>	Search instruments for matching conditions
<code>instfixed</code>	Construct fixed-rate instrument
<code>instfloat</code>	Construct floating-rate instrument
<code>instfloor</code>	Construct floor instrument
<code>instget</code>	Data from instrument variable
<code>instgetcell</code>	Data and context from instrument variable
<code>instlength</code>	Count instruments
<code>instlookback</code>	Construct lookback option
<code>instoptbnd</code>	Construct bond option
<code>instoptstock</code>	Construct stock option
<code>instselect</code>	Create instrument subset by matching conditions
<code>instsetfield</code>	Add or reset data for existing instruments
<code>instswap</code>	Construct swap instrument
<code>insttypes</code>	List types

Financial Object Structures

<code>classfin</code>	Create financial structure or return financial structure class name
<code>isafin</code>	Returns True if input argument is financial structure type or financial object class
<code>stockspec</code>	Create stock structure

Interest Term Structure

date2time	Time and frequency from dates
disc2rate	Interest rates from cash flow discounting factors
intenvget	Properties of interest-rate structure
intenvset	Set properties of interest-rate structure
rate2disc	Discount factors from interest rates
ratetimes	Change time intervals defining interest-rate environment
time2date	Dates from time and frequency

Date

datedisp	Display date entries
----------	----------------------

Graphical Display

treeviewer	Tree information
------------	------------------

Functions — Alphabetical List

asianbycrr

Purpose Price Asian option from CRR binomial tree

Syntax `Price = asianbycrr(CRRTree, OptSpec, Strike, Settle, ExerciseDates, AmericanOpt, AvgType, AvgPrice, AvgDate)`

Arguments

<code>CRRTree</code>	Stock tree structure created by <code>crrtree</code> .
<code>OptSpec</code>	NINST-by-1 list of string values 'Call' or 'Put'.
<code>Strike</code>	NINST-by-1 vector of strike price values. Each row is the schedule for one option.
<code>Settle</code>	NINST-by-1 vector of <code>Settle</code> dates. The settle date for every Asian option is set to the valuation date of the stock tree. The Asian argument <code>Settle</code> is ignored.
<code>ExerciseDates</code>	For a European option (<code>AmericanOpt = 0</code>): NINST-by-1 vector 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 (<code>AmericanOpt = 1</code>): NINST-by-2 vector of exercise date boundaries. For each instrument, the option can be exercised on any tree date between or including the pair of dates on that row. If only one non-NaN date is listed, or if <code>ExerciseDates</code> is NINST-by-1, the option can be exercised between the valuation date of the stock tree and the single listed exercise date.

AmericanOpt	(Optional) If AmericanOpt = 0, NaN, or is unspecified, the option is a European option. If AmericanOpt = 1, the option is an American option.
AvgType	(Optional) String = 'arithmetic' for arithmetic average (default) or 'geometric' for geometric average.
AvgPrice	(Optional) Scalar representing the average price of the underlying asset at Settle. This argument is used when AvgDate < Settle. Default is the current stock price.
AvgDate	(Optional) Scalar representing the date on which the averaging period begins. Default = Settle.

Description

Price = asianbycrr(CRRTree, OptSpec, Strike, Settle, ExerciseDates, AmericanOpt, AvgType, AvgPrice, AvgDate) calculates the value of fixed- and floating-strike Asian options. To compute the value of a floating-strike Asian option, specify Strike as NaN. Fixed-strike Asian options are also known as average price options. Floating-strike Asian options are also known as average strike options.

Price is a NINST-by-1 vector of expected prices at time 0.

Asian options are priced using Hull-White (1993). Consequently, for these options only the root node contains a unique price.

Examples

Price a floating-strike Asian option using a CRR binomial tree.

Load the file deriv.mat, which provides CRRTree. The CRRTree structure contains the stock specification and time information needed to price the option.

```
load deriv.mat;
```

Set the required values. Other arguments will use defaults.

```
OptSpec = 'put';  
Strike = NaN;  
Settle = '01-Jan-2003';  
ExerciseDates = '01-Jan-2004';
```

Use `asianbycrr` to compute the price of the option.

```
Price = asianbycrr(CRRTree, OptSpec, Strike, Settle, ...  
ExerciseDates)
```

```
Price =
```

```
1.2177
```

References

Hull, J., and A. White, "Efficient Procedures for Valuing European and American Path-Dependent Options," *Journal of Derivatives*, Volume 1, pp. 21-31.

See Also

`crrtree`, `instasian`

Purpose Price Asian option from EQP binomial tree

Syntax Price = asianbyeqp(EQPTree, OptSpec, Strike, Settle, ExerciseDates, AmericanOpt, AvgType, AvgPrice, AvgDate)

Arguments

EQPTree	Stock tree structure created by eqptree.
OptSpec	NINST-by-1 list of string values 'Call' or 'Put'.
Strike	NINST-by-1 vector of strike price values. Each row is the schedule for one option.
Settle	NINST-by-1 vector of Settle dates. The settle date for every Asian option is set to the valuation date of the stock tree. The Asian argument Settle is ignored.
ExerciseDates	For a European option (AmericanOpt = 0): NINST-by-1 vector 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 (AmericanOpt = 1): NINST-by-2 vector of exercise date boundaries. For each instrument, the option can be exercised on any tree 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 valuation date of the stock tree and the single listed exercise date.

AmericanOpt	(Optional) If AmericanOpt = 0, NaN, or is unspecified, the option is a European option. If AmericanOpt = 1, the option is an American option.
AvgType	(Optional) String = 'arithmetic' for arithmetic average (default) or 'geometric' for geometric average.
AvgPrice	(Optional) Scalar representing the average price of the underlying asset at Settle. This argument is used when AvgDate < Settle. Default is the current stock price.
AvgDate	(Optional) Scalar representing the date on which the averaging period begins.

Description

Price = asianbyeqp(EQPTree, OptSpec, Strike, Settle, ExerciseDates, AmericanOpt, AvgType, AvgPrice, AvgDate) calculates the value of fixed- and floating-strike Asian options. To compute the value of a floating-strike Asian option, specify Strike as NaN. Fixed-strike Asian options are also known as average price options. Floating-strike Asian options are also known as average strike options.

Price is a NINST-by-1 vector of expected prices at time 0.

Examples

Price a floating-strike Asian option using an EQP equity tree.

Load the file deriv.mat, which provides EQPTree. The EQPTree structure contains the stock specification and time information needed to price the option.

```
load deriv.mat;
```

Set the required values. Other arguments will use defaults.

```
OptSpec = 'put';  
Strike = NaN;
```

```
Settle = '01-Jan-2003';  
ExerciseDates = '01-Jan-2004';
```

Use `asianbyeqp` to compute the price of the option.

```
Price = asianbyeqp(EQPTree, OptSpec, Strike, Settle, ...  
ExerciseDates)
```

```
Price =
```

```
1.2724
```

References

Hull, J., and A. White, “Efficient Procedures for Valuing European and American Path-Dependent Options,” *Journal of Derivatives*, Volume 1, pp. 21-31.

See Also

`eqptree`, `instasian`

Purpose Price Asian options using an implied trinomial tree (ITT)

Syntax Price = asianbyitt(ITTTree, OptSpec, Strike, Settle, ExerciseDates, AmericanOpt, AvgType, AvgPrice, AvgDate)

Arguments

ITTTree	Stock tree structure created by ittree.
OptSpec	NINST-by-1 list of string values 'call' or 'put'.
Strike	NINST-by-1 vector of strike price values. Each row represents the schedule for one option.
Settle	NINST-by-1 vector of Settle dates. The settle date for every Asian option is set to the valuation date of the stock tree. The Asian argument Settle is ignored.
ExerciseDates	For a European option (AmericanOpt = 0): NINST-by-1 vector of exercise dates. Each row is the schedule for one option. For a European option, there is only one exercise date which is the option expiry date. For an American option (AmericanOpt = 1): NINST-by-2 vector of exercise date boundaries. For each instrument, the option can be exercised on any tree 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 valuation date of the stock tree and the single listed exercise date.

AmericanOpt	(Optional) If AmericanOpt = 0, NaN, or is unspecified, the option is a European option. If AmericanOpt = 1, the option is an American option.
AvgType	(Optional) String = 'arithmetic' for arithmetic average (default) or 'geometric' for geometric average.
AvgPrice	(Optional) Scalar representing the average price of the underlying asset at Settle. This argument is used when AvgDate < Settle. Default is the current stock price.
AvgDate	(Optional) Scalar representing the date on which the averaging period begins.

Description

Price = asianbyitt(ITTree, OptSpec, Strike, Settle, ExerciseDates, AmericanOpt, AvgType, AvgPrice, AvgDate) calculates the value of fixed- and floating-strike Asian options. To compute the value of a floating-strike Asian option, specify Strike as NaN. Fixed-strike Asian options are also known as average price options. Floating-strike Asian options are also known as average strike options.

Price is a NINST-by-1 vector of expected prices at time 0.

Note The Settle date for every Asian option is set to the ValuationDate of the stock tree. The Asian argument, Settle, is ignored.

Examples

Price a floating-strike Asian option using an ITT equity tree.

Load the file deriv.mat which provides the ITTree. The ITTree structure contains the stock specification and time information needed to price the option.

```
load deriv.mat;
```

Set the required values. Other arguments will use defaults.

```
OptSpec = 'put';  
Strike = NaN;  
Settle = '01-Jan-2006';  
ExerciseDates = '01-Jan-2007';
```

Use `asianbyitt` to compute the price of the option.

```
Price = asianbyitt(ITTree, OptSpec, Strike, Settle, ExerciseDates)  
  
Price =  
  
1.0778
```

References

Hull, J., and A. White, “Efficient Procedures for Valuing European and American Path-Dependent Options,” *Journal of Derivatives*, Volume 1, 1993, pp. 21-31.

See Also

`instasian`, `itttree`

Purpose Price barrier option from CRR binomial tree

Syntax [Price, PriceTree] = barrierbycrr(CRRTree, OptSpec, Strike, Settle, ExerciseDates, AmericanOpt, BarrierSpec, Barrier, Rebate, Options)

Arguments

CRRTree	Stock tree structure created by crrtree.
OptSpec	NINST-by-1 list of string values 'Call' or 'Put'.
Strike	NINST-by-1 vector of strike price values. Each row is the schedule for one option.
Settle	NINST-by-1 vector of Settle dates. The settle date for every barrier option is set to the valuation date of the stock tree. The barrier argument Settle is ignored.
ExerciseDates	For a European option (AmericanOpt = 0): NINST-by-1 vector 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 (AmericanOpt = 1): NINST-by-2 vector of exercise date boundaries. For each instrument, the option can be exercised on any tree 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 valuation date of the stock tree and the single listed exercise date.

barrierbycrr

AmericanOpt	If AmericanOpt = 0, NaN, or is unspecified, the option is a European option. If AmericanOpt = 1, the option is an American option.
BarrierSpec	List of string values: 'UI': Up Knock In 'UO': Up Knock Out 'DI': Down Knock In 'DO': Down Knock Out
Barrier	Vector of barrier values.
Rebate	(Optional) NINST-by-1 matrix of rebate values. Default = 0. For Knock-in options, the rebate is paid at expiry. For Knock-out options, the rebate is paid when the barrier is reached.
Options	(Optional) Derivatives pricing options structure created with derivset.

See instbarrier for a description of barrier contract arguments.

Description

[Price, PriceTree] = barrierbycrr(CRRTree, OptSpec, Strike, Settle, ExerciseDates, AmericanOpt, BarrierSpec, Barrier, Rebate, Options) computes the price of barrier options using a CRR binomial tree.

Price is a 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 barrier option using a CRR binomial tree.

Load the file deriv.mat, which provides CRRTree. The CRRTree structure contains the stock specification and time information needed to price the option.


```
load deriv.mat;
```

Set the required values. Other arguments will use defaults.

```
OptSpec = 'Call';  
Strike = 105;  
Settle = '01-Jan-2003';  
ExerciseDates = '01-Jan-2006';  
AmericanOpt = 1;  
BarrierSpec = 'UI';  
Barrier = 102;
```

```
Price = barrierbycrr(CRRTree, OptSpec, Strike, Settle, ...  
ExerciseDates, AmericanOpt, BarrierSpec, Barrier)
```

```
Price =
```

```
12.1272
```

References

Derman, E., I. Kani, D. Ergener and I. Bardhan, “Enhanced Numerical Methods for Options with Barriers,” *Financial Analysts Journal*, (Nov. - Dec. 1995), pp. 65-74.

See Also

crrtree, instbarrier

barrierbyeqp

Purpose Price barrier option from EQP binomial tree

Syntax [Price, PriceTree] = barrierbyeqp(EQPtree, OptSpec, Strike, ExerciseDates, AmericanOpt, BarrierSpec, Barrier, Rebate, Options)

Arguments

EQPtree	Stock tree structure created by eqptree.
OptSpec	NINST-by-1 list of string values 'Call' or 'Put'.
Strike	NINST-by-1 vector of strike price values. Each row is the schedule for one option.
Settle	NINST-by-1 vector of Settle dates. The settle date for every barrier option is set to the valuation date of the stock tree. The barrier argument Settle is ignored.
ExerciseDates	For a European option (AmericanOpt = 0): NINST-by-1 vector 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 (AmericanOpt = 1): NINST-by-2 vector of exercise date boundaries. For each instrument, the option can be exercised on any tree 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 valuation date of the stock tree and the single listed exercise date.

AmericanOpt	If AmericanOpt = 0, NaN, or is unspecified, the option is a European option. If AmericanOpt = 1, the option is an American option.
BarrierSpec	List of string values: 'UI': Up Knock In 'UO': Up Knock Out 'DI': Down Knock In 'DO': Down Knock Out
Barrier	Vector of barrier values.
Rebate	(Optional) NINST-by-1 matrix of rebate values. Default = 0. For Knock-in options, the rebate is paid at expiry. For Knock-out options, the rebate is paid when the barrier is reached.
Options	(Optional) Derivatives pricing options structure created with derivset.

See instbarrier for a description of barrier contract arguments.

Description

[Price, PriceTree] = barrierbyeqp(EQPtree, OptSpec, Strike, ExerciseDates, AmericanOpt, BarrierSpec, Barrier, Rebate, Options) computes the price of barrier options using an equal probabilities binomial tree.

Price is a 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 barrier option using an EQP equity tree.

Load the file deriv.mat, which provides EQPtree. The EQPtree structure contains the stock specification and time information needed to price the option.

barrierbyeqp

```
load deriv.mat;
```

Set the required values. Other arguments will use defaults.

```
OptSpec = 'Call';  
Strike = 105;  
Settle = '01-Jan-2003';  
ExerciseDates = '01-Jan-2006';  
AmericanOpt = 1;  
BarrierSpec = 'UI';  
Barrier = 102;  
  
Price = barrierbyeqp(EQPTree, OptSpec, Strike, Settle, ...  
ExerciseDates, AmericanOpt, BarrierSpec, Barrier)  
  
Price =  
  
12.2632
```

References

Derman, E., I. Kani, D. Ergener and I. Bardhan, “Enhanced Numerical Methods for Options with Barriers,” *Financial Analysts Journal*, (Nov. - Dec. 1995), pp. 65-74.

See Also

eqptree, instbarrier

Purpose

Price barrier options using an implied trinomial tree (ITT)

Syntax

```
[Price, PriceTree] = barrierbyitt(ITTree, OptSpec, Strike,
ExerciseDates, AmericanOpt, BarrierSpec, Barrier, Rebate,
Options)
```

Arguments

ITTree	Stock tree structure created by ittree.
OptSpec	NINST-by-1 list of string values 'Call' or 'Put'.
Strike	European and American option, NINST-by-1 vector of strike price values. Each row is the schedule for one option.
Settle	NINST-by-1 vector of Settle dates. The settle date for every barrier option is set to the valuation date of the stock tree. The barrier argument Settle is ignored.
ExerciseDates	For a European option (AmericanOpt = 0): NINST-by-1 vector 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 (AmericanOpt = 1): NINST-by-2 vector of exercise date boundaries. For each instrument, the option can be exercised on any tree 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 valuation date of the stock tree and the single listed exercise date.

barrierbyitt

AmericanOpt	If AmericanOpt = 0, NaN, or is unspecified, the option is a European option. If AmericanOpt = 1, the option is an American option.
BarrierSpec	List of string values: 'UI': Up Knock In 'UO': Up Knock Out 'DI': Down Knock In 'DO': Down Knock Out
Barrier	Vector of barrier values.
Rebate	(Optional) NINST-by-1 matrix of rebate values. Default = 0. For Knock In options, the rebate is paid at expiry. For Knock Out options, the rebate is paid when the barrier is reached.
Options	(Optional) Derivatives pricing options structure created with derivset.

See instbarrier for a description of barrier contract arguments.

Description

[Price, PriceTree] = barrierbyitt(ITTree, OptSpec, Strike, ExerciseDates, AmericanOpt, BarrierSpec, Barrier, Rebate, Options) computes the price of barrier options using an implied trinomial tree.

Price is a NINST-by-1 vector of expected prices at time 0.

PriceTree is a tree structure with a vector of instrument prices at each node.

Note The Settle date for every barrier option is set to the ValuationDate of the stock tree. The barrier argument, Settle, is ignored.

Examples

Price a barrier option using an ITT tree.

Load the file `deriv.mat` which provides the `ITTree`. The `ITTree` structure contains the stock specification and time information needed to price the option.

```
load deriv.mat;
```

Set the required values. Other arguments will use defaults.

```
OptSpec = 'Call';  
Strike = 85;  
Settle = '01-Jan-2006';  
ExerciseDates = '31-Dec-2008';  
AmericanOpt = 1;  
BarrierSpec = 'UI';  
Barrier = 115;
```

```
Price = barrierbyitt(ITTree,OptSpec,Strike,Settle,ExerciseDates,AmericanOpt,  
BarrierSpec,Barrier)
```

```
Price =
```

```
2.407
```

References

Derman, E., I. Kani, D. Ergener, and I. Bardhan, “Enhanced Numerical Methods for Options with Barriers,” *Financial Analysts Journal*, Nov.-Dec.,1995.

See Also

`instbarrier`, `itttree`

bdtprice

Purpose Instrument prices from BDT interest-rate tree

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

Arguments

<code>BDTree</code>	Interest-rate tree structure created by <code>bdttree</code> .
<code>InstSet</code>	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.
<code>Options</code>	(Optional) Derivatives pricing options structure created with <code>derivset</code> .

Description

`[Price, PriceTree] = bdtprice(BDTree, 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 from a BDT tree.
- `capbybdt`: Price a cap from a BDT tree.
- `cfbybdt`: Price an arbitrary set of cash flows from a BDT tree.
- `fixedbybdt`: Price a fixed-rate note from a BDT tree.
- `floatbybdt`: Price a floating-rate note from a BDT tree.
- `floorbybdt`: Price a floor from a BDT tree.
- `optbndbybdt`: Price a bond option from a BDT tree.
- `swapbybdt`: Price a swap from 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);

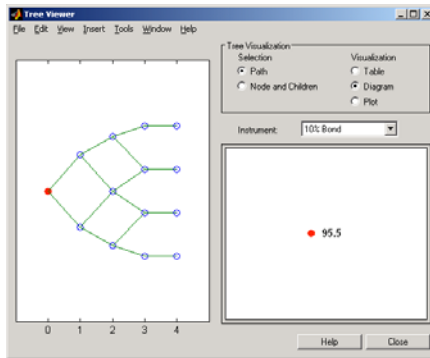
Warning: Not all cash flows are aligned with the tree. Result will
be approximated.

Price =
```

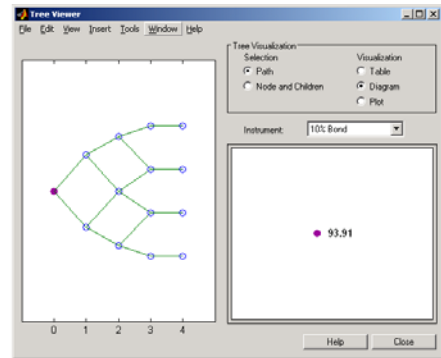
bdtprice

95.5030
93.9079
1.4863

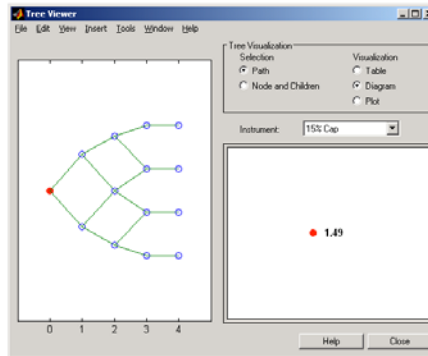
You can use treeviewer to see the prices of these three instruments along the price tree.



First 10% Bond (Maturity 2003)



Second 10% Bond (Maturity 2004)



15% Cap

See Also

bdtrens, bdttree, instadd, intenprice, intenrens

Purpose Instrument prices and sensitivities from 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 the `bdttree` function. 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(BDTree, 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; for example, T = F is one year. Compounding = 365 $Disc = (1 + Z/F)^{-T}$, where F is the number of days in the basis year and T is a number of days elapsed computed by basis. Compounding = -1 $Disc = \exp(-T*Z)$, where T is time in years.

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 five 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'];

TimeSpec = bdttimespec(ValuationDate, Maturity, Compounding)

TimeSpec =

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

See Also

`bdttree`, `bdtvolspec`

bdttree

Purpose Construct BDT interest-rate tree

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

Arguments

<code>VolSpec</code>	Volatility process specification. See <code>bdtvolspec</code> for information on the volatility process.
<code>RateSpec</code>	Interest-rate specification for the initial rate curve. See <code>intenvset</code> for information on declaring an interest-rate variable.
<code>TimeSpec</code>	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 <code>bdttimespec</code> 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,...
```

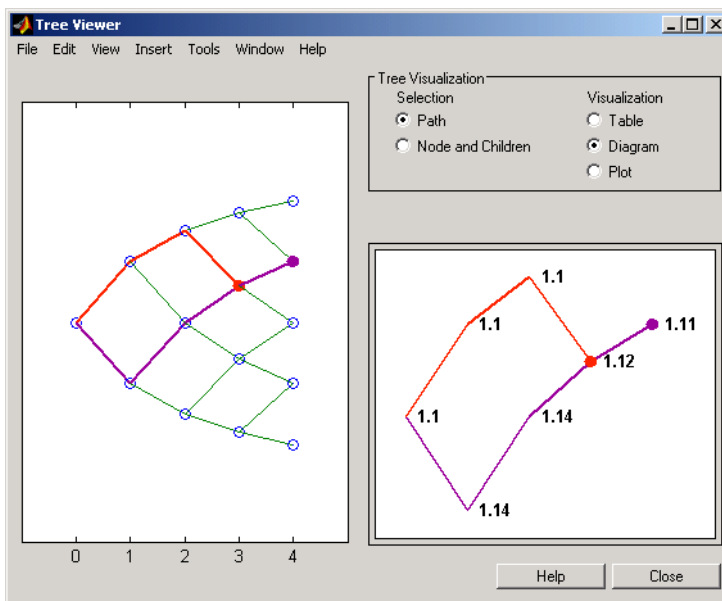


```
'ValuationDate', ValuationDate,...
'StartDates', StartDate,...
'EndDates', EndDates,...
'Rates', Rates);
```

```
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

bdtvolspec

Purpose Specify BDT interest-rate volatility process

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

Arguments

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

bkprice

Purpose Instrument prices from Black-Karasinski interest-rate tree

Syntax `[Price, PriceTree] = bkprice(BKTree, InstSet, Options)`

Arguments

<code>BKTree</code>	Interest-rate tree structure created by <code>bktree</code> .
<code>InstSet</code>	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.
<code>Options</code>	(Optional) Derivatives pricing options structure created with <code>derivset</code> .

Description

`[Price, PriceTree] = bkprice(BKTree, InstSet, Options)` computes arbitrage free prices for instruments using an interest-rate tree created with `bktree`. 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.

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

Related single-type pricing functions are

- `bondbybk`: Price a bond from a Black-Karasinski tree.
- `capbybk`: Price a cap from a Black-Karasinski tree.
- `cfbybk`: Price an arbitrary set of cash flows from a Black-Karasinski tree.
- `fixedbybk`: Price a fixed-rate note from a Black-Karasinski tree.
- `floatbybk`: Price a floating-rate note from a Black-Karasinski tree.
- `floorbybk`: Price a floor from a Black-Karasinski tree.
- `optbndbybk`: Price a bond option from a Black-Karasinski tree.
- `swapbybk`: Price a swap from a Black-Karasinski tree.

Examples

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

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

instdisp(BKSubSet)

Index Type   CouponRate Settle      Maturity   Period Name ...
1      Bond    0.03         01-Jan-2004 01-Jan-2007 1      3% bond
2      Bond    0.03         01-Jan-2004 01-Jan-2008 2      3% bond

Index Type Strike Settle      Maturity   CapReset... Name ...
3      Cap    0.04         01-Jan-2004 01-Jan-2008 1      4% Cap

[Price, PriceTree] = bkprice(BKTree, BKSubSet);

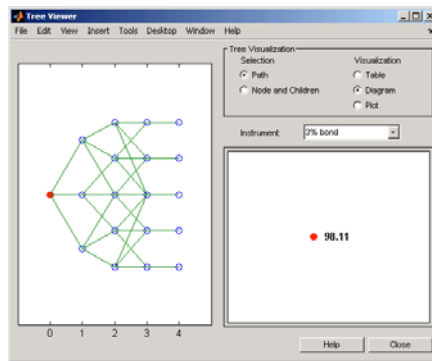
Price =

    98.1096
    95.6734
```

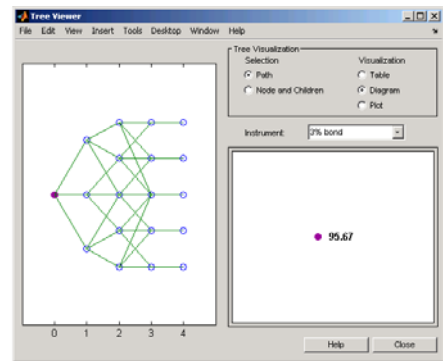
2.2706

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

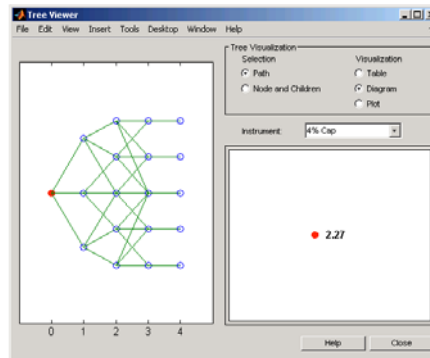
```
treeviewer(PriceTree, BKSubSet)
```



First 3% Bond (Maturity 2007)



Second 3% Bond (Maturity 2008)



4% Cap

See Also

bksens, bktree, instadd, intenprice, intenvsens

Purpose Instrument prices and sensitivities from Black-Karasinski interest-rate tree

Syntax [Delta, Gamma, Vega, Price] = bksens(BKTree, InstSet, Options)

Arguments

BKTree	Interest-rate tree structure created by bktree.
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] = bksens(BKTree, InstSet, Options) computes instrument sensitivities and prices for instruments using an interest-rate tree created with the bktree function. NINST instruments from a financial instrument variable, InstSet, are priced. bksens 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 bktree. See bktree 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 bktree.

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 `bktree`. See `bkvolspec` 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;
BKSubSet = instselect(BKInstSet, 'Type', {'Bond', 'Cap'});

instdisp(BKSubSet)

Index Type CouponRate Settle      Maturity      Period  Name
...
1      Bond 0.03          01-Jan-2004   01-Jan-2007   1       3% Bond
2      Bond 0.03          01-Jan-2004   01-Jan-2008   1       3% Bond
3      Cap 0.04          01-Jan-2004   01-Jan-2008   1       4% Cap

[Index, Gamma] = bksens(BKTree, BKSubSet)

Delta =
```


-285.7151

-365.7048

189.5319

Gamma =

1.0e+003 *

0.8456

1.4345

6.9999

See Also

bkprice, bktree, bkvolspec, instadd

bktimespec

Purpose Specify time structure for Black-Karasinski tree

Syntax TimeSpec = bktimespec(ValuationDate, Maturity, Compounding)

Arguments

ValuationDate Scalar date marking the pricing date and first observation in the tree. Specify as a 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 (continuous compounding). 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; for example, T = F is one year.

Compounding = 365

Disc = $(1 + Z/F)^{-T}$, where F is the number of days in the basis year and T is a number of days elapsed computed by basis.

Compounding = -1

Disc = $\exp(-T*Z)$, where T is time in years.

Description

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

TimeSpec is a structure specifying the time layout for bktree. 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 a four-period tree with annual nodes. Use annual compounding to report rates.

```
ValuationDate = 'Jan-1-2004';  
Maturity = ['12-31-2004'; '12-31-2005'; '12-31-2006';  
           '12-31-2007'];  
Compounding = 1;  
TimeSpec = bktimespec(ValuationDate, Maturity, Compounding)
```

```
TimeSpec =
```

```
    FinObj: 'BKTimeSpec'  
ValuationDate: 731947  
    Maturity: [4x1 double]  
    Compounding: 1  
    Basis: 0
```

See Also

bktree, bkvolspec, hwtree

bktree

Purpose Construct Black-Karasinski interest-rate tree

Syntax `BKTree = bktree(VolSpec, RateSpec, TimeSpec)`

Arguments

<code>VolSpec</code>	Volatility process specification. See <code>bkvolspec</code> for information on the volatility process.
<code>RateSpec</code>	Interest-rate specification for the initial rate curve. See <code>intenvset</code> for information on declaring an interest-rate variable.
<code>TimeSpec</code>	Tree time layout specification. Defines the observation dates of the BK tree and the Compounding rule for date to time mapping and price-yield formulas. See <code>bktimespec</code> for information on the tree structure.

Description `BKTree = bktree(VolSpec, RateSpec, TimeSpec)` creates a structure containing time and interest-rate information on a recombining tree.

Examples Using the data provided, create a BK volatility specification (`VolSpec`), rate specification (`RateSpec`), and tree time layout specification (`TimeSpec`). Then use these specifications to create a BK tree using `bktree`.

```
Compounding = -1;
ValuationDate = '01-01-2004';
StartDate = ValuationDate;
VolDates = ['12-31-2004'; '12-31-2005'; '12-31-2006';
'12-31-2007'];
VolCurve = 0.01;
AlphaDates = '01-01-2008';
AlphaCurve = 0.1;
Rates = [0.0275; 0.0312; 0.0363; 0.0415];
```

```

BKVolSpec = bkvolspec(ValuationDate, VolDates, VolCurve,...
AlphaDates, AlphaCurve);

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

BKTimeSpec = bktimespec(ValuationDate, VolDates, Compounding);

BKTree = bktree(BKVolSpec, RateSpec, BKTimeSpec)

BKTree =

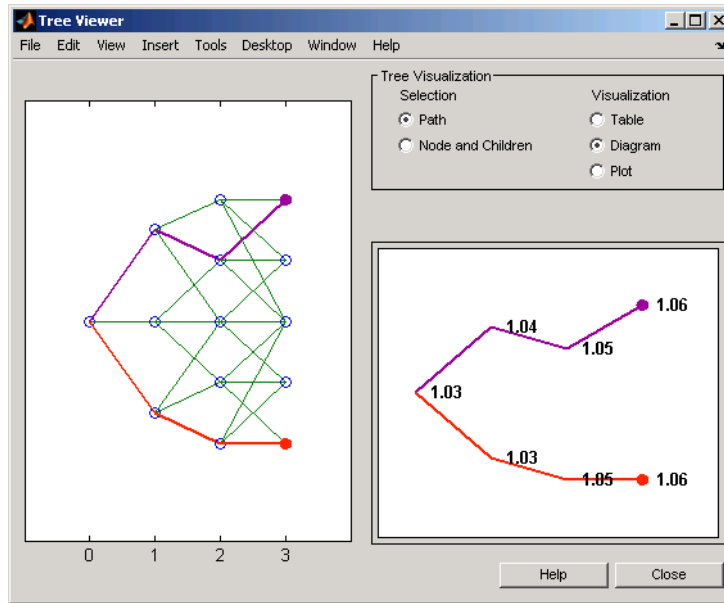
    FinObj: 'BKFwdTree'
    VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
    tObs: [0 0.9973 1.9973 2.9973]
    dObs: [731947 732312 732677 733042]
    CFlowT: {[4x1 double] [3x1 double] [2x1 double] [3.9973]}
    Probs: {[3x1 double] [3x3 double] [3x5 double]}
    Connect: {[2] [2 3 4] [2 2 3 4 4]}
    FwdTree: {1x4 cell}

```

Use `treeview` to observe the tree you have created.

```
treeview(BKTree)
```

bktree



See Also `bkprice`, `bktimespec`, `bkvolspec`, `intenvset`

Purpose Specify Black-Karasinski interest-rate volatility process

Syntax `Volspec = bkvolspec(ValuationDate, VolDates, VolCurve, AlphaDates, AlphaCurve, 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.
AlphaDates	NPOINTS-by-1 vector of mean reversion end dates.
AlphaCurve	NPOINTS-by-1 vector of positive mean reversion values in decimal form.
InterpMethod	(Optional) Interpolation method. Default is 'linear'. See <code>interp1</code> for more information.

Description `Volspec = bkvolspec(ValuationDate, VolDates, VolCurve, AlphaDates, AlphaCurve, InterpMethod)` creates a structure specifying the volatility for `bktree`.

Examples Using the data provided, create a Black-Karasinski volatility specification (`Volspec`).

```
ValuationDate = '01-01-2004';
StartDate = ValuationDate;
VolDates = ['12-31-2004'; '12-31-2005'; '12-31-2006';
'12-31-2007'];
VolCurve = 0.01;
AlphaDates = '01-01-2008';
```

```
AlphaCurve = 0.1;  
BKVolSpec = bkvolspec(ValuationDate, VolDates, VolCurve,...  
AlphaDates, AlphaCurve)
```

```
BKVolSpec =  
  
    FinObj: 'BKVolSpec'  
ValuationDate: 731947  
    VolDates: [4x1 double]  
    VolCurve: [4x1 double]  
    AlphaCurve: 0.1000  
    AlphaDates: 733408  
VolInterpMethod: 'linear'
```

See Also

bktree, interp1

Purpose Price bond from BDT interest-rate tree

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

Arguments

BDTree	Interest-rate tree structure created by <code>bdttree</code> .
CouponRate	Decimal annual rate.
Settle	Settlement date. A vector of serial date numbers or date strings. <code>Settle</code> must be earlier than or equal to <code>Maturity</code> .
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 instrument. A vector of integers. <ul style="list-style-type: none">• 0 = actual/actual (default)• 1 = 30/360 (SIA)• 2 = actual/360• 3 = actual/365• 4 = 30/360 (PSA)• 5 = 30/360 (ISDA)• 6 = 30/360 (European)• 7 = actual/365 (Japanese)

	<ul style="list-style-type: none">• 8 = actual/actual (ISMA)• 9 = actual/360 (ISMA)• 10 = actual/365 (ISMA)
EndMonthRule	(Optional) End-of-month rule. A vector. This rule applies only when <code>Maturity</code> 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 <code>FirstCouponDate</code> and <code>LastCouponDate</code> are both specified, <code>FirstCouponDate</code> 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 <code>FirstCouponDate</code> , a specified <code>LastCouponDate</code> determines the coupon structure of the bond. The coupon structure of a bond is truncated at the <code>LastCouponDate</code> 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 <code>derivset</code> .

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(BDTree, CouponRate, Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate, StartDate, Face, Options)` computes the price of a bond from 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 `BDTree`. `BDTree` contains the time and interest-rate information needed to price the bond.

```
load deriv.mat;
```

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(BDTree, CouponRate, Settle, Maturity, Period)
```

```
Price =
```

bondbybdt

95.5030

See Also

bdttree, bdtprice, instbond

Purpose

Price bond from Black-Karasinski interest-rate tree

Syntax

```
[Price, PriceTree] = bondbybk(BKTree, CouponRate,  
Settle, Maturity, Period, Basis, EndMonthRule, IssueDate,  
FirstCouponDate, LastCouponDate, StartDate, Face, Options)
```

Arguments

BKTree	Forward rate tree structure created by <code>bktree</code> .
CouponRate	Decimal annual rate.
Settle	Settlement date. A vector of serial date numbers or date strings. <code>Settle</code> must be earlier than or equal to <code>Maturity</code> .
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 instrument. A vector of integers. <ul style="list-style-type: none">• 0 = actual/actual (default)• 1 = 30/360 (SIA)• 2 = actual/360• 3 = actual/365• 4 = 30/360 (PSA)• 5 = 30/360 (ISDA)• 6 = 30/360 (European)• 7 = actual/365 (Japanese)

	<ul style="list-style-type: none">• 8 = actual/actual (ISMA)• 9 = actual/360 (ISMA)• 10 = actual/365 (ISMA)
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 BK tree. The bond argument `Settle` is ignored.

Description

`[Price, PriceTree] = bondbybk(BKTree, CouponRate, Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate, StartDate, Face, Options)` computes the price of a bond from a Black-Karasinski interest-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.PTree` contains the clean prices.
- `PriceTree.AITree` contains the accrued interest.
- `PriceTree.tObs` contains the observation times.

Examples

Price a 4% bond using a Black-Karasinski interest-rate tree.

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

```
load deriv.mat;
```

Set the required values. Other arguments will use defaults.

```
CouponRate = 0.04;
Settle = '01-Jan-2004';
Maturity = '31-Dec-2008';
```

Use `bondbybk` to compute the price of the bond.

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

Price =

98.0300

See Also

bkprice, bktree, hwprice, hwtree, instbond

Purpose

Price bond from 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 <code>hjmtree</code> .
CouponRate	Decimal annual rate.
Settle	Settlement date. A vector of serial date numbers or date strings. <code>Settle</code> must be earlier than or equal to <code>Maturity</code> .
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 instrument. A vector of integers. <ul style="list-style-type: none">• 0 = actual/actual (default)• 1 = 30/360 (SIA)• 2 = actual/360• 3 = actual/365• 4 = 30/360 (PSA)• 5 = 30/360 (ISDA)• 6 = 30/360 (European)• 7 = actual/365 (Japanese)

	<ul style="list-style-type: none">• 8 = actual/actual (ISMA)• 9 = actual/360 (ISMA)• 10 = actual/365 (ISMA)
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 from 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`. The `HJMTree` structure contains the time and forward-rate information needed to price the bond.

```
load deriv.mat;
```

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

Purpose

Price bond from Hull-White interest-rate tree

Syntax

```
[Price, PriceTree] = bondbyhw(HWTree, CouponRate,  
Settle, Maturity, Period, Basis, EndMonthRule, IssueDate,  
FirstCouponDate, LastCouponDate, StartDate, Face, Options)
```

Arguments

HWTree	Forward-rate tree structure created by hwtree.
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 instrument. A vector of integers. <ul style="list-style-type: none">• 0 = actual/actual (default)• 1 = 30/360 (SIA)• 2 = actual/360• 3 = actual/365• 4 = 30/360 (PSA)• 5 = 30/360 (ISDA)• 6 = 30/360 (European)• 7 = actual/365 (Japanese)

	<ul style="list-style-type: none">• 8 = actual/actual (ISMA)• 9 = actual/360 (ISMA)• 10 = actual/365 (ISMA)
EndMonthRule	(Optional) End-of-month rule. A vector. This rule applies only when <code>Maturity</code> 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 <code>FirstCouponDate</code> and <code>LastCouponDate</code> are both specified, <code>FirstCouponDate</code> 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 <code>FirstCouponDate</code> , a specified <code>LastCouponDate</code> determines the coupon structure of the bond. The coupon structure of a bond is truncated at the <code>LastCouponDate</code> 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 <code>derivset</code> .

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

Description

`[Price, PriceTree] = bondbyhw(HWTree, CouponRate, Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate, StartDate, Face, Options)` computes the price of a bond from a Hull-White interest-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.PTree` contains the clean prices.
- `PriceTree.AITree` contains the accrued interest.
- `PriceTree.tObs` contains the observation times.

Examples

Price a 4% bond using a Hull-White interest-rate tree.

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

```
load deriv.mat;
```

Set the required values. Other arguments will use defaults.

```
CouponRate = 0.04;
Settle = '01-Jan-2004';
Maturity = '31-Dec-2008';
```

Use `bondbyhw` to compute the price of the bond.

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

Price =

98.0483

See Also

bkprice, bktree, hwprice, hwtree, instbond

Purpose Price bond from set of zero curves

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

Arguments

RateSpec	Structure containing the properties of an interest-rate structure. See <code>intenvset</code> for information on creating RateSpec.
CouponRate	Decimal annual rate.
Settle	Settlement date. A vector of serial date numbers or date strings. <code>Settle</code> must be earlier than or equal to <code>Maturity</code> .
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 instrument. A vector of integers. <ul style="list-style-type: none">• 0 = actual/actual (default)• 1 = 30/360 (SIA)• 2 = actual/360• 3 = actual/365• 4 = 30/360 (PSA)• 5 = 30/360 (ISDA)• 6 = 30/360 (European)• 7 = actual/365 (Japanese)

	<ul style="list-style-type: none">• 8 = actual/actual (ISMA)• 9 = actual/360 (ISMA)• 10 = actual/365 (ISMA)
EndMonthRule	(Optional) End-of-month rule. A vector. This rule applies only when <code>Maturity</code> 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 <code>FirstCouponDate</code> and <code>LastCouponDate</code> are both specified, <code>FirstCouponDate</code> 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 <code>FirstCouponDate</code> , a specified <code>LastCouponDate</code> determines the coupon structure of the bond. The coupon structure of a bond is truncated at the <code>LastCouponDate</code> 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.mat;
```

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

bushpath

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])
```

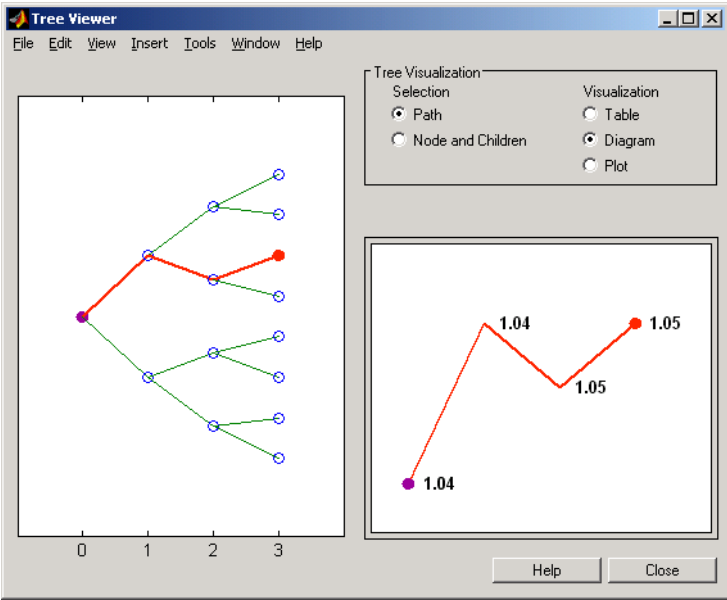
returns the rates at the tree nodes located by taking the up branch, then the down branch, and finally the up branch again.

```
FwdRates =  
  
    1.0356  
    1.0364  
    1.0526
```

1.0463

You can visualize this with the treeviewer function.

```
treeviewer(HJMTree)
```



See Also bushshape, mkbush

bushshape

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.

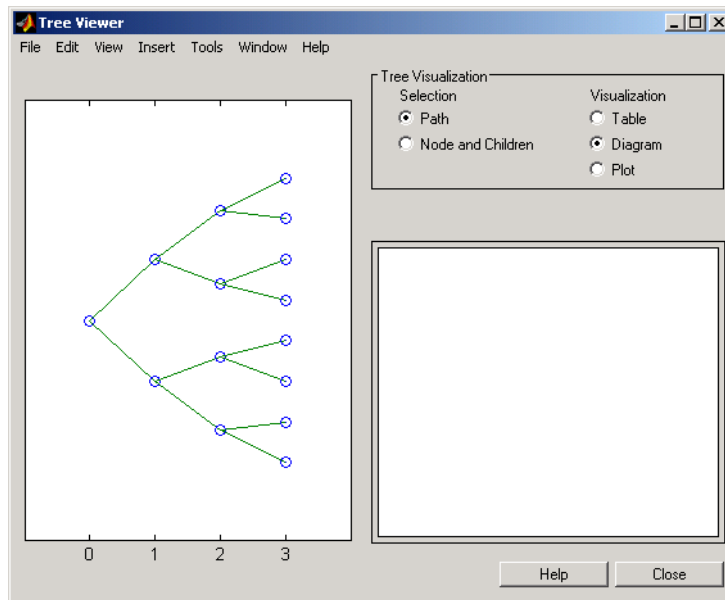
Trim is 1 if NumPos decreases by 1 when moving from one time level to the next. Otherwise, it is 0.

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.



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
```

bushshape

```
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`

Purpose Price cap instrument from BDT interest-rate tree

Syntax [Price, PriceTree] = capbybdt(BDTree, Strike, Settle, Maturity, Reset, Basis, Principal, Options)

Arguments

BDTree	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) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none">• 0 = actual/actual (default)• 1 = 30/360 (SIA)• 2 = actual/360• 3 = actual/365• 4 = 30/360 (PSA)• 5 = 30/360 (ISDA)• 6 = 30/360 (European)• 7 = actual/365 (Japanese)• 8 = actual/actual (ISMA)• 9 = actual/360 (ISMA)

- 10 = actual/365 (ISMA)
- Principal (Optional) The notional principal amount. Default = 100.
- Options (Optional) Derivatives pricing options structure created with derivset.

Description

[Price, PriceTree] = capbybdt(BDTree, 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 BDTree. The BDTree structure contains the time and interest-rate information needed to price the cap instrument.

```
load deriv.mat;
```

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(BDTree, Strike, Settle, Maturity)  
  
Price =
```

28.5191

Example 2. This example shows 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. Remaining arguments will use defaults.

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

Use capbybdt to find the price of the cap instrument.

capbybdt

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

```
Price =
```

```
1.6923
```

See Also

bdttree, cfbybdt, floorbybdt, swapbybdt

Purpose Price cap instrument from Black-Karasinski interest-rate tree

Syntax [Price, PriceTree] = capbybk(BKTree, Strike, Settle, Maturity, Reset, Basis, Principal, Options)

Arguments

BKTree	Interest-rate tree structure created by bktree.
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) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none">• 0 = actual/actual (default)• 1 = 30/360 (SIA)• 2 = actual/360• 3 = actual/365• 4 = 30/360 (PSA)• 5 = 30/360 (ISDA)• 6 = 30/360 (European)• 7 = actual/365 (Japanese)• 8 = actual/actual (ISMA)• 9 = actual/360 (ISMA)

- 10 = actual/365 (ISMA)
- Principal (Optional) The notional principal amount. Default = 100.
- Options (Optional) Derivatives pricing options structure created with derivset.

Description

[Price, PriceTree] = capbybk(BKTree, Strike, Settle, Maturity, Reset, Basis, Principal, Options) computes the price of a cap instrument from a Black-Karasinski 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 BK tree. The cap argument Settle is ignored.

Examples

Price a 3% cap instrument using a Black-Karasinski interest-rate tree.

Load the file deriv.mat, which provides BKTree. The BKTree structure contains the time and interest-rate information needed to price the cap instrument.

```
load deriv.mat;
```

Set the required values. Other arguments will use defaults.

```
Strike = 0.03;  
Settle = '01-Jan-2005';  
Maturity = '01-Jan-2009';
```

Use capbybk to compute the price of the cap instrument.

```
Price = capbybk(BKTree, Strike, Settle, Maturity)  
  
Price =
```

6.8337

See Also

cfbybk, floorbybk, bktree, swapbybk

Purpose Price cap instrument from HJM interest-rate tree

Syntax `[Price, PriceTree] = capbyhjm(HJMTree, Strike, Settle, Maturity, Reset, Basis, Principal, Options)`

Arguments

HJMTree	Forward-rate tree structure created by <code>hjmtree</code> .
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) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none">• 0 = actual/actual (default)• 1 = 30/360 (SIA)• 2 = actual/360• 3 = actual/365• 4 = 30/360 (PSA)• 5 = 30/360 (ISDA)• 6 = 30/360 (European)• 7 = actual/365 (Japanese)• 8 = actual/actual (ISMA)• 9 = actual/360 (ISMA)

- 10 = actual/365 (ISMA)
- 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`. The `HJMTree` structure contains the time and forward-rate information needed to price the cap instrument.

```
load deriv.mat;
```

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 =
```

capbyhjm

6.2831

See Also

cfbyhjm, floorbyhjm, hjmtree, swapbyhjm

Purpose

Price cap instrument from Hull-White interest-rate tree

Syntax

[Price, PriceTree] = capbyhw(HWTtree, Strike, Settle, Maturity, Reset, Basis, Principal, Options)

Arguments

HWTtree	Interest-rate tree structure created by hwtree.
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) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none">• 0 = actual/actual (default)• 1 = 30/360 (SIA)• 2 = actual/360• 3 = actual/365• 4 = 30/360 (PSA)• 5 = 30/360 (ISDA)• 6 = 30/360 (European)• 7 = actual/365 (Japanese)• 8 = actual/actual (ISMA)• 9 = actual/360 (ISMA)

- 10 = actual/365 (ISMA)
- Principal (Optional) The notional principal amount. Default = 100.
- Options (Optional) Derivatives pricing options structure created with derivset.

Description

[Price, PriceTree] = capbyhw(HWTree, Strike, Settle, Maturity, Reset, Basis, Principal, Options) computes the price of a cap instrument from a Hull-White 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 HW tree. The cap argument Settle is ignored.

Examples

Price a 3% cap instrument using a Hull-White interest-rate tree.

Load the file deriv.mat, which provides HWTree. The HWTree structure contains the time and interest-rate information needed to price the cap instrument.

```
load deriv.mat;
```

Set the required values. Other arguments will use defaults.

```
Strike = 0.03;  
Settle = '01-Jan-2005';  
Maturity = '01-Jan-2009';
```

Use capbyhw to compute the price of the cap instrument.

```
Price = capbyhw(HWTree, Strike, Settle, Maturity)  
  
Price =
```

7.0707

See Also

cfbyhw, floorbyhw, hwtree, swapbyhw

Purpose Price cash flows from BDT interest-rate tree

Syntax `[Price, PriceTree] = cfbybdt(BDTTree, CFlowAmounts, CFlowDates, Settle, Basis, Options)`

Arguments

<code>BDTTree</code>	Forward-rate tree structure created by <code>bdttree</code> .
<code>CFlowAmounts</code>	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.
<code>CFlowDates</code>	NINST-by-MOSTCFS matrix of cash flow dates. Each entry contains the serial date number of the corresponding cash flow in <code>CFlowAmounts</code> .
<code>Settle</code>	Settlement date. A vector of serial date numbers or date strings. The <code>Settle</code> date for every cash flow is set to the <code>ValuationDate</code> of the BDT tree. The cash flow argument, <code>Settle</code> , is ignored.
<code>Basis</code>	(Optional) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none">• 0 = actual/actual (default)• 1 = 30/360 (SIA)• 2 = actual/360• 3 = actual/365• 4 = 30/360 (PSA)• 5 = 30/360 (ISDA)• 6 = 30/360 (European)

- 7 = actual/365 (Japanese)
- 8 = actual/actual (ISMA)
- 9 = actual/360 (ISMA)
- 10 = actual/365 (ISMA)

Options (Optional) Derivatives pricing options structure created with derivset.

Description

`[Price, PriceTree] = cfbybdt(BDTree, 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 `BDTree`. The `BDTree` structure contains the time and interest-rate information needed to price the instruments.

```
load deriv.mat;
```

The valuation date (settle date) specified in `BDTree` is January 1, 2000 (date number 730486).

```
BDTree.RateSpec.ValuationDate
```

```
ans =
```

```
730486
```

Provide values for the other required arguments.

```
CFlowAmounts =[5 NaN 5.5 105; 5 0 6 105];  
CFlowDates = [730852, NaN, 731582, 731947;  
              730852, 731217, 731582, 731947];
```

Use this information to compute the prices of the two cash flow instruments.

```
[Price, PriceTree] = cfbybdt(BDTree, CFlowAmounts, ...  
CFlowDates, BDTree.RateSpec.ValuationDate)
```

```
Price =
```

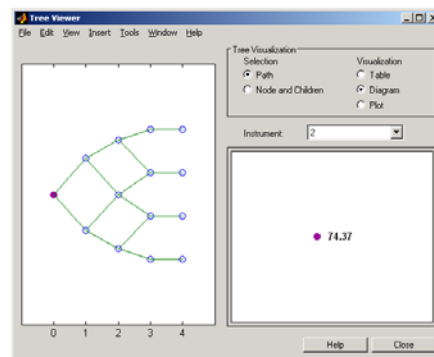
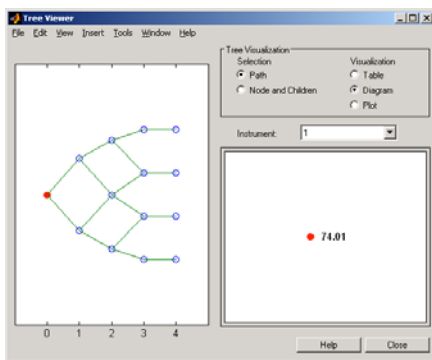
```
    74.0112  
    74.3671
```

```
PriceTree =
```

```
FinObj: 'BDTreePriceTree'  
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 Black-Karasinski interest-rate tree

Syntax [Price, PriceTree] = cfbybk(BKTree, CFlowAmounts, CFlowDates, Settle, Basis, Options)

Arguments

BKTree	Forward-rate tree structure created by bktree.
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 serial date number 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 BK tree. The cash flow argument, Settle, is ignored.
Basis	(Optional) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none">• 0 = actual/actual (default)• 1 = 30/360 (SIA)• 2 = actual/360• 3 = actual/365• 4 = 30/360 (PSA)• 5 = 30/360 (ISDA)• 6 = 30/360 (European)

- 7 = actual/365 (Japanese)
- 8 = actual/actual (ISMA)
- 9 = actual/360 (ISMA)
- 10 = actual/365 (ISMA)

Options (Optional) Derivatives pricing options structure created with derivset.

Description

`[Price, PriceTree] = cfbybk(BKTree, CFlowAmounts, CFlowDates, Settle, Basis, Options)` prices cash flows from a Black-Karasinski 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, 2005 to January 1, 2009.

Load the file `deriv.mat`, which provides `BKTree`. The `BKTree` structure contains the time and interest-rate information needed to price the instruments.

```
load deriv.mat;
```

The valuation date (settle date) specified in `BKTree` is January 1, 2004 (date number 731947).

```
BKTree.RateSpec.ValuationDate
```

```
ans =
```

```
731947
```

Provide values for the other required arguments.

```
CFlowAmounts =[5 NaN 5.5 105; 5 0 6 105];  
CFlowDates = [732678, NaN, 733408,733774;  
              732678, 733034, 733408, 734774];
```

Use this information to compute the prices of the two cash flow instruments.

```
[Price, PriceTree] = cfbybk(BKTree, CFlowAmounts, CFlowDates,...  
BKTree.RateSpec.ValuationDate)
```

```
Price =
```

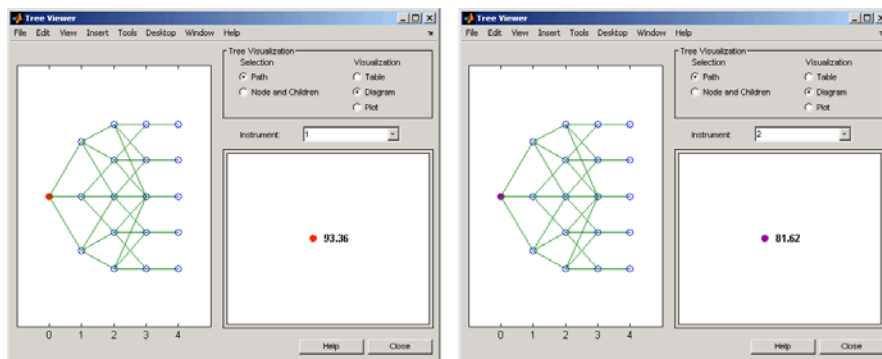
```
93.3600  
81.6218
```

```
PriceTree =
```

```
FinObj: 'BKPriceTree'  
tObs: [0 1 2 3 4]  
PTree: {[2x1 double] [2x3 double] [2x5 double] [2x5  
double] [2x5 double]}  
Connect: {[2] [2 3 4] [2 2 3 4 4]}  
Probs: {[3x1 double] [3x3 double] [3x5 double]}
```

You can visualize the prices of the two cash flow instruments with the `treeviewer` function.

```
treeviewer(PriceTree)
```

**See Also**

bktree, bkprice, 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 serial date number 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 instrument. A vector of integers. <ul style="list-style-type: none">• 0 = actual/actual (default)• 1 = 30/360 (SIA)• 2 = actual/360• 3 = actual/365• 4 = 30/360 (PSA)• 5 = 30/360 (ISDA)• 6 = 30/360 (European)

- 7 = actual/365 (Japanese)
- 8 = actual/actual (ISMA)
- 9 = actual/360 (ISMA)
- 10 = actual/365 (ISMA)

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`. The `HJMTree` structure contains the time and forward-rate information needed to price the instruments.

```
load deriv.mat;
```

The valuation date (settle date) specified in `HJMTree` is January 1, 2000 (date number 730486).

```
HWTree.RateSpec.ValuationDate
```

```
ans =
```

```
730486
```

Provide values for the other required arguments.

```
CFlowAmounts =[5 NaN 5.5 105; 5 0 6 105];  
CFlowDates = [730852, NaN, 731582, 731947;  
              730852, 731217, 731582, 731947];
```

Use this information to compute the prices of the two cash flow instruments.

```
[Price, PriceTree] = cfbyhjm(HJMTree, CFlowAmounts,...  
                             CFlowDates, HJMTree.RateSpec.ValuationDate)
```

```
Price =
```

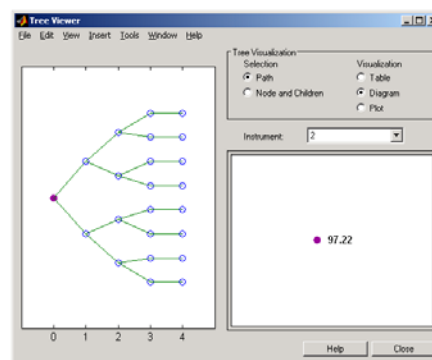
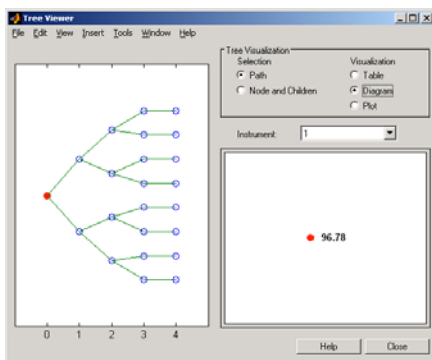
```
    96.7805  
    97.2188
```

```
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 from Hull-White interest-rate tree

Syntax `[Price, PriceTree] = cfbyhw(HWTree, CFlowAmounts, CFlowDates, Settle, Basis, Options)`

Arguments

HWTree	Forward-rate tree structure created by <code>hwtree</code> .
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 serial date number 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 HW tree. The cash flow argument, Settle, is ignored.
Basis	(Optional) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none">• 0 = actual/actual (default)• 1 = 30/360 (SIA)• 2 = actual/360• 3 = actual/365• 4 = 30/360 (PSA)• 5 = 30/360 (ISDA)• 6 = 30/360 (European)

- 7 = actual/365 (Japanese)
- 8 = actual/actual (ISMA)
- 9 = actual/360 (ISMA)
- 10 = actual/365 (ISMA)

Options (Optional) Derivatives pricing options structure created with derivset.

Description

`[Price, PriceTree] = cfbyhw(HWTree, CFlowAmounts, CFlowDates, Settle, Basis, Options)` prices cash flows from a Hull-White 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, 2005 to January 1, 2009.

Load the file `deriv.mat`, which provides `HWTree`. The `HWTree` structure contains the time and interest-rate information needed to price the instruments.

```
load deriv.mat;
```

The valuation date (settle date) specified in `HWTree` is January 1, 2004 (date number 731947).

```
HWTree.RateSpec.ValuationDate
```

```
ans =
```

```
731947
```

Provide values for the other required arguments.

```
CFlowAmounts =[5 NaN 5.5 105; 5 0 6 105];  
CFlowDates = [732678, NaN, 733408, 733774;  
              732678, 733034, 733408, 734774];
```

Use this information to compute the prices of the two cash flow instruments.

```
[Price, PriceTree] = cfbyhw(HWTree, CFlowAmounts, CFlowDates,...  
HWTree.RateSpec.ValuationDate)
```

```
Price =
```

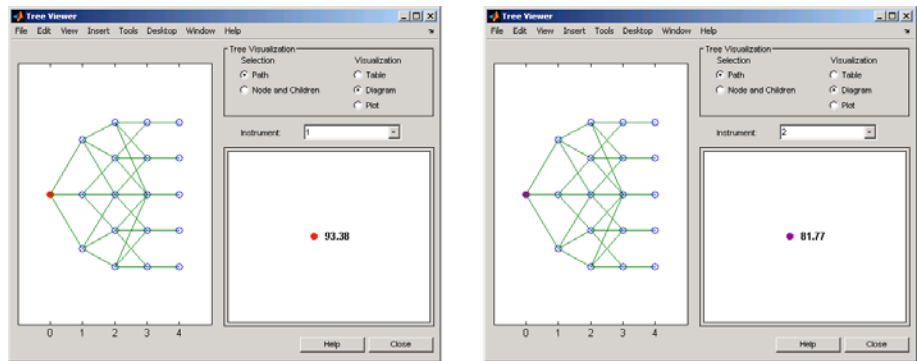
```
93.3789  
81.7651
```

```
PriceTree =
```

```
FinObj: 'HWPriceTree'  
tObs: [0 1 2 3 4]  
PTree: {[2x1 double] [2x3 double] [2x5 double] [2x5  
double] [2x5 double]}  
Connect: {[2] [2 3 4] [2 2 3 4 4]}  
Probs: {[3x1 double] [3x3 double] [3x5 double]}
```

You can visualize the prices of the two cash flow instruments with the `treeviewer` function.

```
treeviewer(PriceTree)
```

**See Also**

cfamounts, hwtree, hwprice, instcf

Purpose Price cash flows from set of zero curves

Syntax `Price = cfbyzero(RateSpec, CFlowAmounts, CFlowDates, Settle, Basis)`

Arguments

RateSpec	Structure containing the properties of an interest-rate structure. See <code>intenvset</code> 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 instrument. A vector of integers. <ul style="list-style-type: none">• 0 = actual/actual (default)• 1 = 30/360 (SIA)• 2 = actual/360• 3 = actual/365• 4 = 30/360 (PSA)• 5 = 30/360 (ISDA)• 6 = 30/360 (European)

- 7 = actual/365 (Japanese)
- 8 = actual/actual (ISMA)
- 9 = actual/360 (ISMA)
- 10 = actual/365 (ISMA)

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`. The `ZeroRateSpec` structure contains the interest-rate information needed to price the instruments.

```
load deriv.mat
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`

classfin

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 the name of a 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

compoundbycrr

Purpose Price compound option from CRR binomial tree

Syntax [Price, PriceTree] = compoundbycrr(CRRTree, UOptSpec, UStrike, USettle, UExerciseDates, UAmericanOpt, COptSpec, CStrike, CSettle, CExerciseDates, CAmericanOpt)

Arguments

CRRTree	Stock tree structure created by crrtree.
UOptSpec	String = 'Call' or 'Put'.
UStrike	1-by-1 vector of strike price values.
USettle	1-by-1 vector of Settle dates.
UExerciseDates	For a European option (UAmericanOpt = 0): 1-by-1 vector of exercise dates. For a European option, there is only one exercise date, the option expiry date. For an American option (UAmericanOpt = 1): 1-by-2 vector of exercise date boundaries. The option can be exercised on any tree date. If only one non-NaN date is listed, or if ExerciseDates is 1-by-1, the option can be exercised between the valuation date of the stock tree and the single listed exercise date.
UAmericanOpt	If UAmericanOpt = 0, NaN, or is unspecified, the option is a European option. If UAmericanOpt = 1, the option is an American option.
COptSpec	NINST-by-1 list of string values 'Call' or 'Put' of the compound option.
CStrike	NINST-by-1 vector of strike price values. Each row is the schedule for one option.

CSettle	1-by-1 vector containing the settlement or trade date.
CExerciseDates	For a European option (CAmericanOpt = 0): NINST-by-1 vector 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 (CAmericanOpt = 1): NINST-by-2 vector of exercise date boundaries. For each instrument, the option can be exercised on any tree 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 valuation date of the stock tree and the single listed exercise date.
CAmericanOpt	(Optional) If CAmericanOpt = 0, NaN, or is unspecified, the option is a European option. If CAmericanOpt = 1, the option is an American option.

Description

[Price, PriceTree] = compoundbycrr(CRRTree, UOptSpec, UStrike, USettle, UExerciseDates, UAmericanOpt, COptSpec, CStrike, CSettle, CExerciseDates, CAmericanOpt) calculates the value of a compound option.

Price is a 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 compound option using a CRR binomial tree.

Load the file `deriv.mat`, which provides `CRRTree`. The `CRRTree` structure contains the stock specification and time information needed to price the option.

```
load deriv.mat
```

Set the required values. Other arguments will use defaults.

```
UOptSpec = 'Call';
UStrike = 130;
USettle = '01-Jan-2003';
UExerciseDates = '01-Jan-2006';
UAmericanOpt = 1;
COptSpec = 'Put';
CStrike = 5;
CSettle = '01-Jan-2003';
CExerciseDates = '01-Jan-2005';

Price = compoundbycrr(CRRTree, UOptSpec, UStrike, USettle, ...
    UExerciseDates, UAmericanOpt, COptSpec, CStrike, CSettle, ...
    CExerciseDates)

Price =

    2.8482
```

References

Rubinstein, Mark, "Double Trouble," *Risk* 5, 1991, p. 73.

See Also

`crrtree`, `instcompound`

Purpose

Price compound option from EQP binomial tree

Syntax

```
[Price, PriceTree] = compoundbyeqp(EQPtree, UOptSpec, UStrike,
    USettle, UExerciseDates, UAmericanOpt, COptSpec,
    CStrike, CSettle, CExerciseDates, CAmericanOpt)
```

Arguments

EQPtree	Stock tree structure created by eqptree.
UOptSpec	String = 'Call' or 'Put'.
UStrike	1-by-1 vector of strike price values.
USettle	1-by-1 vector of Settle dates.
UExerciseDates	For a European option (UAmericanOpt = 0): 1-by-1 vector of exercise dates. For a European option, there is only one exercise date, the option expiry date. For an American option (UAmericanOpt = 1): 1-by-2 vector of exercise date boundaries. The option can be exercised on any tree date. If only one non-NaN date is listed, or if ExerciseDates is 1-by-1, the option can be exercised between the valuation date of the stock tree and the single listed exercise date.
UAmericanOpt	If UAmericanOpt = 0, NaN, or is unspecified, the option is a European option. If UAmericanOpt = 1, the option is an American option.
COptSpec	NINST-by-1 list of string values 'Call' or 'Put' of the compound option.
CStrike	NINST-by-1 vector of strike price values. Each row is the schedule for one option.

CSettle	1-by-1 vector containing the settlement or trade date.
CExerciseDates	For a European option (CAmericanOpt = 0): NINST-by-1 vector 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 (CAmericanOpt = 1): NINST-by-2 vector of exercise date boundaries. For each instrument, the option can be exercised on any tree 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 valuation date of the stock tree and the single listed exercise date.
CAmericanOpt	If CAmericanOpt = 0, NaN, or is unspecified, the option is a European option. If CAmericanOpt = 1, the option is an American option.

Description

[Price, PriceTree] = compoundbyeqp(EQPTree, UOptSpec, UStrike, USettle, UExerciseDates, UAmericanOpt, COptSpec, CStrike, CSettle, CExerciseDates, CAmericanOpt) calculates the value of a compound option.

Price is a 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 compound option using an EQP equity tree.

Load the file `deriv.mat`, which provides `EQPTree`. The `EQPTree` structure contains the stock specification and time information needed to price the option.

```
load deriv.mat
```

Set the required values. Other arguments will use defaults.

```
UOptSpec = 'Call';
UStrike = 130;
USettle = '01-Jan-2003';
UExerciseDates = '01-Jan-2006';
UAmericanOpt = 1;
COptSpec = 'Put';
CStrike = 5;
CSettle = '01-Jan-2003';
CExerciseDates = '01-Jan-2005';

Price = compoundbyeqp(EQPTree, UOptSpec, UStrike, USettle, ...
    UExerciseDates, UAmericanOpt, COptSpec, CStrike, CSettle, ...
    CExerciseDates)

Price =

    3.3931
```

References

Rubinstein, Mark, "Double Trouble," *Risk* 5, 1991, p. 73

See Also

`eqptree`, `instcompound`

compoundbyitt

Purpose Price compound options using an implied trinomial tree (ITT)

Syntax [Price, PriceTree] = compoundbyitt(ITTTree, UOptSpec, UStrike, USettle, UExerciseDates, UAmericanOpt, COptSpec, CStrike, CSettle, CExerciseDates, CAmericanOpt)

Arguments

ITTTree	Stock tree structure created by itttree.
UOptSpec	String = 'call' or 'put'.
UStrike	1-by-1 vector of strike price values.
USettle	1-by-1 vector of Settle dates.
UExerciseDates	For a European option (UAmericanOpt = 0): 1-by-1 vector of exercise dates. For a European option, there is only one exercise date, the option expiry date. For an American option (UAmericanOpt = 1): 1-by-2 vector of exercise date boundaries. The option can be exercised on any tree date. If only one non-NaN date is listed, or if ExerciseDates is 1-by-1, the option can be exercised between the valuation date of the stock tree and the single listed exercise date.
UAmericanOpt	If UAmericanOpt = 0, NaN, or is unspecified, the option is a European option. If UAmericanOpt = 1, the option is an American option.
COptSpec	NINST-by-1 list of string values 'Call' or 'Put' of the compound option.
CStrike	NINST-by-1 vector of strike price values. Each row is the schedule for one option.

CSettle	1-by-1 vector containing the settlement or trade date.
CExerciseDates	For a European option (CAmericanOpt = 0): NINST-by-1 vector 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 (CAmericanOpt = 1): NINST-by-2 vector of exercise date boundaries. For each instrument, the option can be exercised on any tree 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 valuation date of the stock tree and the single listed exercise date.
CAmericanOpt	(Optional) If CAmericanOpt = 0, NaN, or is unspecified, the option is a European option. If CAmericanOpt = 1, the option is an American option.

Description

[Price, PriceTree] = compoundbyitt(ITTree, UOptSpec, UStrike, USettle, UExerciseDates, UAmericanOpt, COptSpec, CStrike, CSettle, CExerciseDates, CAmericanOpt) calculates the value of a compound option by an ITT trinomial tree.

Price is a NINST-by-1 vector of expected prices at time 0.

PriceTree is a tree structure with a vector of instrument prices at each node.

Note The Settle date is set to the ValuationDate of the stock tree.

Examples

Price a compound option using an ITT tree.

Load the file `deriv.mat` which provides the `ITTTree`. The `ITTTree` structure contains the stock specification and time information needed to price the option.

```
load deriv.mat
```

Set the required values. Other arguments will use defaults.

```
UOptSpec = 'Call';
UStrike = 99;
USettle = '01-Jan-2006';
UExerciseDates = '01-Jan-2010';
UAmericanOpt = 1;
COptSpec = 'Put';
CStrike = 5;
CSettle = '01-Jan-2006';
CExerciseDates = '01-Jan-2010';

Price = compoundbyitt(ITTTree, UOptSpec, UStrike, USettle, ...
    UExerciseDates, UAmericanOpt, COptSpec, CStrike, CSettle, ...
    CExerciseDates)

Price =

    2.727
```

References

Rubinstein, Mark, "Double Trouble," *Risk* 5, 1991.

See Also

`instcompound`, `itttree`

Purpose Instrument prices from CRR tree

Syntax [Price, PriceTree] = crrprice(CRRTree, InstSet, Options)

Arguments

CRRTree	Interest-rate tree structure created by <code>crrtree</code> .
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 <code>derivset</code> .

Description

[Price, PriceTree] = crrprice(CRRTree, InstSet, Options) computes stock option prices using a CRR binomial tree created with `crrtree`.

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 stock tree. If an instrument cannot be priced, NaN is returned.

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

PriceTree.PTree contains the prices.

PriceTree.tObs contains the observation times.

PriceTree.dObs contains the observation dates.

crrprice handles instrument types: 'Asian', 'Barrier', 'Compound', 'Lookback', 'OptStock'. See `instadd` to construct defined types.

Related single-type pricing functions are

- `asianbycrr`: Price an Asian option from a CRR tree.

- `barrierbycrr`: Price a barrier option from a CRR tree.
- `compoundbycrr`: Price a compound option from a CRR tree.
- `lookbackbycrr`: Price a lookback option from a CRR tree.
- `optstockbycrr`: Price an American, Bermuda, or European option from a CRR tree.

Examples

Load the CRR tree and instruments from the data file `deriv.mat`. Price the barrier and lookback options contained in the instrument set.

```
load deriv.mat;
CRRSubSet = instselect(CRRInstSet,'Type', ...
{'Barrier', 'Lookback'});

instdisp(CRRSubSet)

Index Type OptSpec Strike Settle ExerciseDates AmericanOpt BarrierSpec ...
1 Barrier call 105 01-Jan-2003 01-Jan-2006 1 ui ...

Index Type OptSpec Strike Settle ExerciseDates AmericanOpt Name Quantity
2 Lookback call 115 01-Jan-2003 01-Jan-2006 0 Lookback1 7
3 Lookback call 115 01-Jan-2003 01-Jan-2007 0 Lookback2 9

[Price, PriceTree] = crrprice(CRRTree, CRRSubSet)

Price =

    12.1272
     7.6015
    11.7772

PriceTree =

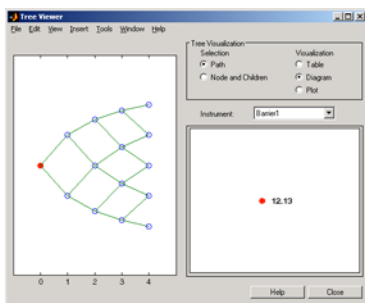
    FinObj: 'BinPriceTree'
    PTree: {1x5 cell}
```

tObs: [0 1 2 3 4]

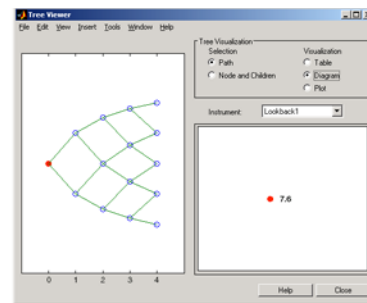
dObs: [731582 731947 732313 732678 733043]

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

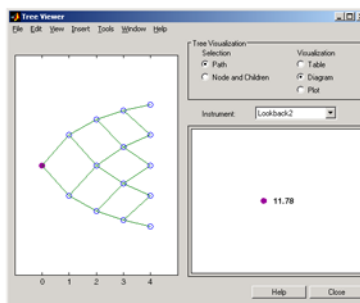
```
treeviewer(PriceTree, CRRSubSet)
```



Barrier1



Lookback1



Lookback2

See Also

crrsens, crrtree, instadd

Purpose Instrument prices and sensitivities from CRR tree

Syntax `[Delta, Gamma, Vega, Price] = crrsens(BDTree, InstSet, Options)`

Arguments

<code>CRRTree</code>	Interest-rate tree structure created by <code>crrtree</code> .
<code>InstSet</code>	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.
<code>Options</code>	(Optional) Derivatives pricing options structure created with <code>derivset</code> .

Description

`[Delta, Gamma, Vega, Price] = crrsens(BDTree, InstSet, Options)` computes dollar sensitivities and prices for instruments using a binomial tree created with `crrtree`. NINST instruments from a financial instrument variable, `InstSet`, are priced. `crrsens` handles instrument types: 'Asian', 'Barrier', 'Compound', 'Lookback', 'OptStock'. 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 stock price. Delta is computed by finite differences in calls to `crrtree`. See `crrtree` for information on the stock tree.

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

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 of the stock. Vega is computed by finite differences in calls to `crrtree`.

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

Examples

Load the CRR tree and instruments from the data file `deriv.mat`. Compute the Delta and Gamma sensitivities of the barrier and lookback options contained in the instrument set.

```
load deriv.mat;
CRRSubSet = instselect(CRRInstSet,'Type', ...
{'Barrier', 'Lookback'});

instdisp(CRRSubSet)

Index Type OptSpec Strike Settle ExerciseDates AmericanOpt BarrierSpec ...
1 Barrier call 105 01-Jan-2003 01-Jan-2006 1 ui ...

Index Type OptSpec Strike Settle ExerciseDates AmericanOpt Name Quantity
2 Lookback call 115 01-Jan-2003 01-Jan-2006 0 Lookback1 7
3 Lookback call 115 01-Jan-2003 01-Jan-2007 0 Lookback2 9

[Delta, Gamma] = crrsens(CRRTree, CRRSubSet)

Delta =

0.6885
0.6049
0.8187

Gamma =

0.0310
-0.0000
0.0000
```

crrsens

See Also

crrprice, crrtree

Purpose Specify time structure for CRR tree

Syntax `TimeSpec = crrtimespec(ValuationDate, Maturity, NumPeriods)`

Arguments

<code>ValuationDate</code>	Scalar date indicating the pricing date and first observation in the tree. A serial date number or date string.
<code>Maturity</code>	Scalar date indicating depth of the tree.
<code>NumPeriods</code>	Scalar determining number of time steps in the tree.

Description

`TimeSpec = crrtimespec(ValuationDate, Maturity, NumPeriods)` sets the number of levels and node times for a CRR binomial tree.

`TimeSpec` is a structure specifying the time layout for a CRR binomial tree.

Examples

Specify a four-period CRR tree with time steps of one year.

```
ValuationDate = '1-July-2002';  
Maturity = '1-July-2006';  
TimeSpec = crrtimespec(ValuationDate, Maturity, 4)
```

```
TimeSpec =  
  
    FinObj: 'BinTimeSpec'  
    ValuationDate: 731398  
    Maturity: 732859  
    NumPeriods: 4  
    Basis: 0  
    EndMonthRule: 1  
    tObs: [0 1 2 3 4]  
    dObs: [1x5 double]
```

crrtimespec

See Also

crrtree, stockspec

Purpose Construct CRR stock tree

Syntax `CRRTree = crrtree(StockSpec, RateSpec, TimeSpec)`

Arguments

StockSpec	Stock specification. See <code>stockspec</code> for information on creating a stock specification.
RateSpec	Interest-rate specification for the initial risk free rate curve. See <code>intenvset</code> for information on declaring an interest-rate variable.
TimeSpec	Tree time layout specification. Defines the observation dates of the CRR binomial tree. See <code>crrtimespec</code> for information on the tree structure.

Note The standard CRR tree assumes a constant interest rate, but `RateSpec` allows you to specify an interest-rate curve with varying rates. If you specify variable interest rates, the resulting tree will not be a standard CRR tree.

Description `CRRTree = crrtree(StockSpec, RateSpec, TimeSpec)` creates a structure specifying the time layout for a CRR binomial tree.

Examples Using the data provided, create a stock specification (`StockSpec`), rate specification (`RateSpec`), and tree time layout specification (`TimeSpec`). Then use these specifications to create a CRR tree with `crrtree`.

```
Sigma = 0.20;  
AssetPrice = 50;  
DividendType = 'cash';  
DividendAmounts = [0.50; 0.50; 0.50; 0.50];  
ExDividendDates = {'03-Jan-2003'; '01-Apr-2003'; '05-July-2003';
```

```
'01-Oct-2003'}

StockSpec = stockspec(Sigma, AssetPrice, DividendType, ...
DividendAmounts, ExDividendDates)

StockSpec =

    FinObj: 'StockSpec'
    Sigma: 0.2000
    AssetPrice: 50
    DividendType: 'cash'
    DividendAmounts: [4x1 double]
    ExDividendDates: [4x1 double]

RateSpec = intenvset('Rates', 0.05, 'StartDates',...
'01-Jan-2003', 'EndDates', '31-Dec-2003')

RateSpec =

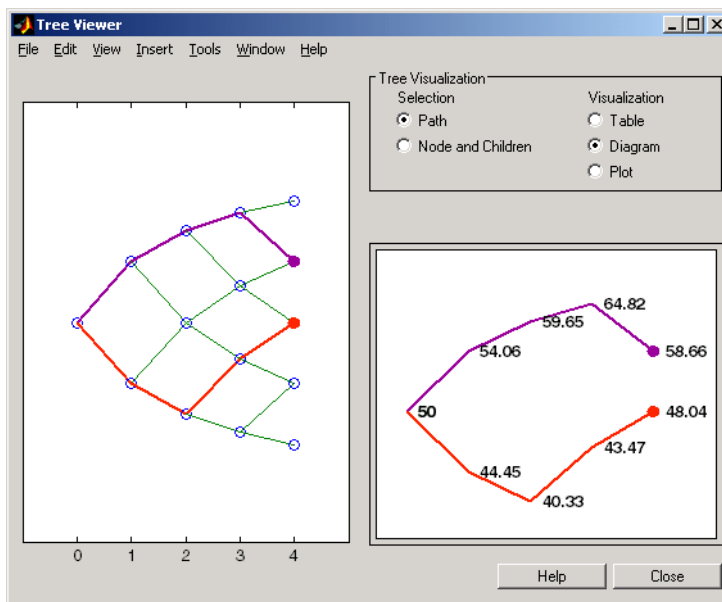
    FinObj: 'RateSpec'
    Compounding: 2
    Disc: 0.9519
    Rates: 0.0500
    EndTimes: 1.9945
    StartTimes: 0
    EndDates: 731946
    StartDates: 731582
    ValuationDate: 731582
    Basis: 0
    EndMonthRule: 1

ValuationDate = '1-Jan-2003';
Maturity = '31-Dec-2003';
TimeSpec = crrtimespec(ValuationDate, Maturity, 4)

TimeSpec =
```

```
        FinObj: 'BinTimeSpec'  
ValuationDate: 731582  
        Maturity: 731946  
        NumPeriods: 4  
        Basis: 0  
        EndMonthRule: 1  
  
CRRTree = crrtree(StockSpec, RateSpec, TimeSpec)  
  
CRRTree =  
  
        FinObj: 'BinStockTree'  
        Method: 'CRR'  
StockSpec: [1x1 struct]  
TimeSpec: [1x1 struct]  
RateSpec: [1x1 struct]  
        tObs: [0 0.2493 0.4986 0.7479 0.9972]  
        dObs: [731582 731672 731763 731856 731946]  
        STree: {1x5 cell}  
        UpProbs: [0.5370 0.5370 0.5370 0.5370]
```

Use `treeview` to observe the tree you have created.



See Also `crrtimespec`, `intenvset`, `stockspec`

Purpose Convert inverse-discount tree to interest-rate tree

Syntax RateTree = cvtree(Tree)

Arguments

Tree Heath-Jarrow-Morton, Black-Derman-Toy, Hull-White, or Black-Karasinski tree structure using inverse-discount notation for forward rates.

Description RateTree = cvtree(Tree) converts a tree structure using inverse-discount notation to a tree structure using rate notation for forward rates.

Examples Convert a Hull-White tree using inverse-discount notation to a Hull-White tree displaying interest-rate notation.

```
load deriv.mat;

HWTTree

HWTTree =

    FinObj: 'HWFwdTree'
    VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
    tObs: [0 1 2 3]
    dObs: [731947 732313 732678 733043]
    CFlowT: {[4x1 double] [3x1 double] [2x1 double] [4]}
    Probs: {[3x1 double] [3x3 double] [3x5 double]}
    Connect: {[2] [2 3 4] [2 2 3 4 4]}
    FwdTree: {1x4 cell}

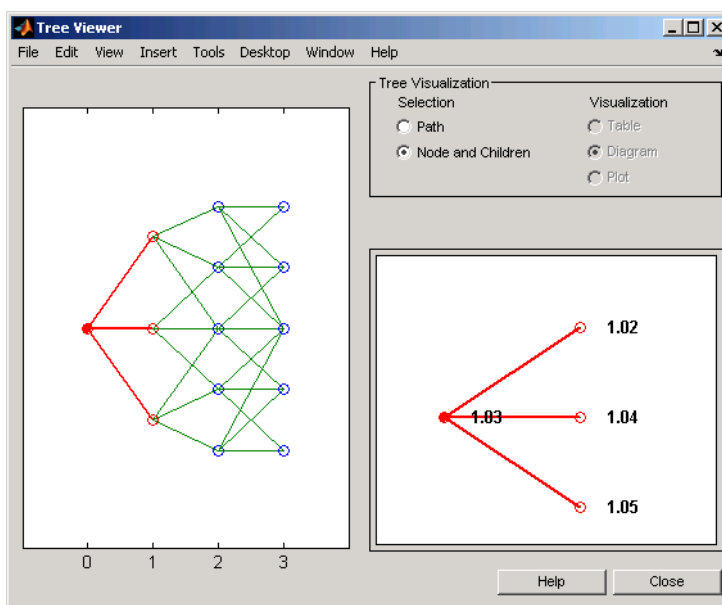
HWTTree.FwdTree{1}
```

cvtree

```
ans =  
    1.0279  
  
HWTree.FwdTree{2}  
  
ans =  
    1.0528    1.0356    1.0186
```

Use `treeviewer` to display the path of interest rates expressed in inverse-discount notation.

```
treeviewer(HWTree)
```



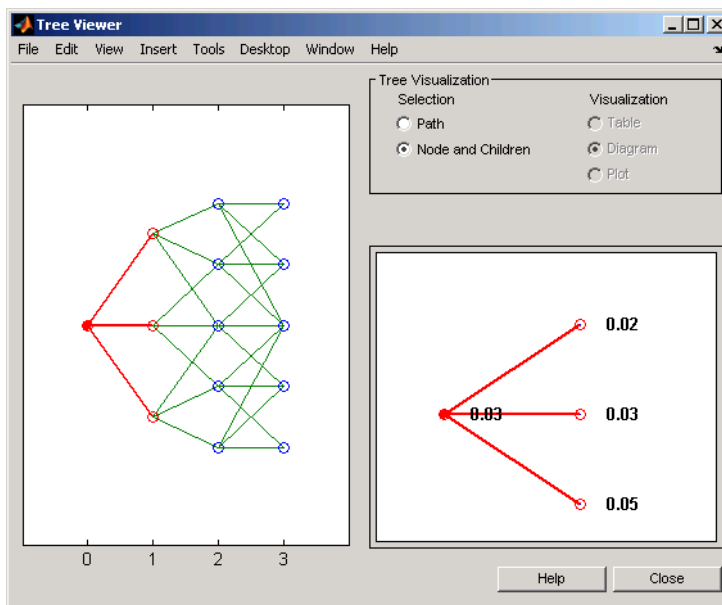
Use `cvtree` to convert the inverse-discount notation to interest-rate notation.

```
RTree = cvtree(HWTree)
```



```
RTree =  
  
    FinObj: 'HWRateTree'  
    VolSpec: [1x1 struct]  
    TimeSpec: [1x1 struct]  
    RateSpec: [1x1 struct]  
    tObs: [0 1 2 3]  
    dObs: [731947 732313 732678 733043]  
    CFlowT: {[4x1 double] [3x1 double] [2x1 double] [4]}  
    Probs: {[3x1 double] [3x3 double] [3x5 double]}  
    Connect: {[2] [2 3 4] [2 2 3 4 4]}  
    RateTree: {1x4 cell}  
  
RTree.RateTree{1}  
  
ans =  
    0.0275  
  
RTree.RateTree{2}  
  
ans =  
    0.0514    0.0349    0.0185
```

Now use `treeview` to display the converted tree, showing the path of interest rates expressed as forward rates.



See Also `disc2rate`, `rate2disc`

Purpose Time and frequency from dates

Syntax [Times, F] = date2time(Settle, Dates, Compounding, Basis, EndMonthRule)

Arguments

Settle	Settlement date. A vector of serial date numbers or date strings.
Dates	Vector of dates corresponding to the compounding value.
Compounding	<p>(Optional) Scalar value representing the rate at which the input zero rates were compounded when annualized. This argument determines the formula for the discount factors:</p> <p>Compounding = 1, 2, 3, 4, 6, 12 (Default = 2.)</p> <p>Disc = $(1 + Z/F)^{-T}$, where F is the compounding frequency, Z is the zero rate, and T is the time in periodic units; for example, T = F is one year.</p> <p>Compounding = 365</p> <p>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.</p> <p>Compounding = -1</p> <p>Disc = $\exp(-T*Z)$, where T is time in years.</p>

Basis	<p>(Optional) Day-count basis of the instrument. A vector of integers.</p> <ul style="list-style-type: none">• 0 = actual/actual (default)• 1 = 30/360 (SIA)• 2 = actual/360• 3 = actual/365• 4 = 30/360 (PSA)• 5 = 30/360 (ISDA)• 6 = 30/360 (European)• 7 = actual/365 (Japanese)• 8 = actual/actual (ISMA)• 9 = actual/360 (ISMA)• 10 = actual/365 (ISMA)
EndMonthRule	<p>(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.</p>

Description

[Times, F] = date2time(Settle, Dates, Compounding, Basis, EndMonthRule) computes time factors appropriate to compounded rate quotes beyond the settlement date.

Times is a vector of time factors.

F is a scalar of related compounding frequencies.

Note To obtain accurate results from this function, the `Basis` and `Dates` arguments must be consistent. If the `Dates` argument contains months that have 31 days, `Basis` must be one of the values that allow months to contain more than 30 days; for example, `Basis = 0, 3, or 7`.

`date2time` is the inverse of `time2date`.

See Also

`cftimes` in Financial Toolbox documentation
`disc2rate`, `rate2disc`, `time2date`

datedisp

Purpose Display date entries

Syntax `datedisp(NumMat, DateForm)`
`CharMat = datedisp(NumMat, DateForm)`

Arguments

`NumMat` Numeric matrix to display.
`DateForm` (Optional) Date format. See `datestr` for available and default format flags.

Description

`datedisp(NumMat, DateForm)` displays 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 (`CharMat = datedisp(NumMat, DateForm)`).

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 Financial Toolbox documentation

Remarks This function is identical to the `datedisp` function in Financial Toolbox.

derivget

Purpose Get derivatives pricing options

Syntax `Value = derivget(Options, 'Parameter')`

Arguments

<i>Options</i>	Existing options specification structure, probably created from previous call to <code>derivset</code> .
<i>Parameter</i>	Must be 'Diagnostics', 'Warnings', 'ConstRate', or 'BarrierMethod'. 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', except for 'BarrierMethod', which can be 'unenhanced' or 'interp'. Specifying 'unenhanced' uses no correction calculation. Specifying 'interp' uses an enhanced valuation interpolating between nodes on barrier boundaries.

Examples Example 1. 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
```

Example 2. 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'.

Example 3. Find the value of 'BarrierMethod' in this structure.

```
derivget(Options, 'BarrierMethod')
```

```
ans =
```

```
unenanced
```

See Also

`barrierbycrr`, `barrierbyeqp`, `derivset`

derivset

Purpose Set or modify derivatives pricing options

Syntax

```
Options = derivset(Options, 'Parameter1', Value1,  
... 'Parameter4', Value4)  
Options = derivset(OldOptions, NewOptions)  
Options = derivset  
derivset
```

Arguments

- | | |
|---------------|---|
| Options | (Optional) Existing options specification structure, probably created from a previous call to <code>derivset</code> . |
| Parameter n | The parameter must be 'Diagnostics', 'Warnings', 'ConstRate', or 'BarrierMethod'. Parameters can be entered in any order. |
| Value n | (BDT, BK, HJM, or HW pricing only) The parameter values for the following three options can be 'on' or 'off': <ul style="list-style-type: none">• '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. |

For pricing barrier options, the 'BarrierMethod' pricing option can be 'unenhanced' (default) or 'interp'. Specifying 'unenhanced' uses no correction calculation. Specifying 'interp' uses an enhanced valuation interpolating between nodes on barrier boundaries.

OldOptions Existing options specification structure.
NewOptions New options specification structure.

Description

`Options = derivset(Options, 'Parameter1', Value1, ... 'Parameter4', Value4)` 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.

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.

derivset

```
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  ]
    BarrierMethod: [ {unenhanced} | interp  ]
```

See Also

barrierbycrr, barrierbyeqp, derivget

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, Basis, EndMonthRule)

Usage 2: ValuationDate is passed and interval points are input as dates.

[Rates, EndTimes, StartTimes] = disc2rate(Compounding, Disc, EndDates, StartDates, ValuationDate, Basis, EndMonthRule)

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; for example, T = F is one year.

Compounding = 365

$Disc = (1 + Z/F)^{-T}$, where F is the number of days in the basis year and T is a number of days elapsed computed by basis.

Compounding = -1

$Disc = \exp(-T*Z)$, where T is time in years.

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.
Basis	(Optional) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none">• 0 = actual/actual (default)• 1 = 30/360 (SIA)• 2 = actual/360• 3 = actual/365• 4 = 30/360 (PSA)• 5 = 30/360 (ISDA)• 6 = 30/360 (European)• 7 = actual/365 (Japanese)• 8 = actual/actual (ISMA)• 9 = actual/360 (ISMA)

- 10 = actual/365 (ISMA)
- EndMonthRule (Optional) End-of-month rule. A vector. This rule applies only when Maturity is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond's coupon payment date is always the same numerical day of the month. 1 = set rule on (default), meaning that a bond's coupon payment date is always the last actual day of the month.

Description

Rates = disc2rate(Compounding, Disc, EndTimes, StartTimes, Basis, EndMonthRule) and [Rates, EndTimes, StartTimes] = disc2rate(Compounding, Disc, EndDates, StartDates, ValuationDate, Basis, EndMonthRule) 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 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, T_{semi} , by the formula $T = T_{\text{semi}}/2 * F$, where F is the compounding frequency.

Specify the investment intervals with either input times (Usage 1) or input dates (Usage 2). Entering ValuationDate invokes the date

disc2rate

interpretation; omitting `ValuationDate` invokes the default time interpretations.

See Also

`rate2disc`, `ratetimes`

Purpose Instrument prices from EQP binomial tree

Syntax [Price, PriceTree] = eqpprice(EQPtree, InstSet, Options)

Arguments

EQPtree	Interest-rate tree structure created by eqptree.
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] = eqpprice(EQPtree, InstSet, Options) computes stock option prices using an EQP binomial tree created with eqptree.

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 stock tree. If an instrument cannot be priced, NaN is returned.

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

PriceTree.PTree contains the prices.

PriceTree.tObs contains the observation times.

PriceTree.dObs contains the observation dates.

eqpprice handles instrument types: 'Asian', 'Barrier', 'Compound', 'Lookback', 'OptStock'. See instadd to construct defined types.

Related single-type pricing functions are

- asianbyeqp: Price an Asian option from an EQP tree.

- barrierbyeqp: Price a barrier option from an EQP tree.
- compoundbyeqp: Price a compound option from an EQP tree.
- lookbackbyeqp: Price a lookback option from an EQP tree.
- optstockbyeqp: Price an American, Bermuda, or European option from an EQP tree.

Examples

Load the EQP tree and instruments from the data file `deriv.mat`. Price the put options contained in the instrument set.

```
load deriv.mat;
EQPSubSet = instselect(EQPInstSet, 'FieldName', 'OptSpec', ...
'Data', 'put')

instdisp(EQPSubSet)

Index Type      OptSpec Strike Settle      ExerciseDates AmericanOpt Name...
1      OptStock put      105  01-Jan-2003 01-Jan-2006      0          Put 105...

Index Type      OptSpec Strike Settle      ExerciseDates AmericanOpt AvgType...
2      Asian put      110  01-Jan-2003 01-Jan-2006      0          arithmetic...
3      Asian put      110  01-Jan-2003 01-Jan-2007      0          arithmetic...

[Price, PriceTree] = eqpprice(EQPTree, EQPSubSet)

Price =

    2.6375
    4.7444
    3.9178

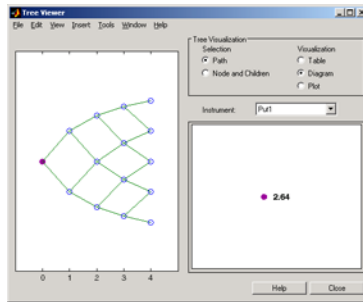
PriceTree =

    FinObj: 'BinPriceTree'
    PTree: {1x5 cell}
```

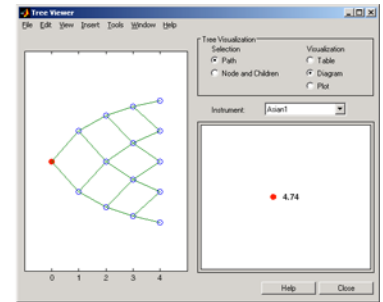
```
tObs: [0 1 2 3 4]
dObs: [731582 731947 732313 732678 733043]
```

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

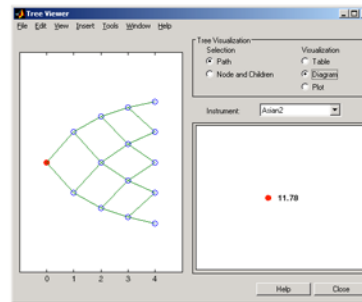
```
treeviewer(PriceTree, EQPSubSet)
```



Put1



Asian1



Asian2

See Also eqpsens, eqptimespec, eqptree

Purpose Instrument prices and sensitivities from EQP binomial tree

Syntax `[Delta, Gamma, Vega, Price] = eqpsens(EQPTree, InstSet, Options)`

Arguments

EQPTree	Interest-rate tree structure created by eqptree.
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] = eqpsens(EQPTree, InstSet, Options)` computes dollar sensitivities and prices for instruments using a binomial tree created with eqptree. NINST instruments from a financial instrument variable, InstSet, are priced. eqpsens handles instrument types: 'Asian', 'Barrier', 'Compound', 'Lookback', 'OptStock'. 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 stock price. Delta is computed by finite differences in calls to eqptree. See eqptree for information on the stock tree.

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

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 of the stock. Vega is computed by finite differences in calls to eqptree.

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

Examples

Load the EQP tree and instruments from the data file `deriv.mat`. Compute the Delta and Gamma sensitivities of the put options contained in the instrument set.

```
load deriv.mat;

EQPSubSet = instselect(EQPInstSet, 'FieldName', 'OptSpec', ...
'Data', 'put')

instdisp(EQPSubSet)

Index Type      OptSpec Strike Settle      ExerciseDates AmericanOpt Name...
1   OptStock put    105   01-Jan-2003 01-Jan-2006      0          Put 105...
```

```
Index Type OptSpec Strike Settle      ExerciseDates AmericanOpt AvgType...
2   Asian put    110   01-Jan-2003 01-Jan-2006      0          arithmetic...
3   Asian put    110   01-Jan-2003 01-Jan-2007      0          arithmetic...
```

```
[Delta, Gamma] = eqpsens(EQPtree, EQPSubSet)

Delta =

-0.2336
-0.5443
-0.4516

Gamma =

0.0218
0.0000
0.0000
```

eqpsens

See Also

eqpprice, eqptree

Purpose Specify time structure for EQP binomial tree

Syntax TimeSpec = eqptimespec(ValuationDate, Maturity, NumPeriods)

Arguments

ValuationDate	Scalar date indicating the pricing date and first observation in the tree. A serial date number or date string.
Maturity	Scalar date indicating depth of the tree.
NumPeriods	Scalar determining number of time steps in the tree.

Description

TimeSpec = eqptimespec(ValuationDate, Maturity, NumPeriods) sets the number of levels and node times for an equal probabilities tree.

TimeSpec is a structure specifying the time layout for an equal probabilities tree.

Examples

Specify a four period tree with time steps of one year.

```
ValuationDate = '1-July-2002';
Maturity = '1-July-2006';
TimeSpec = eqptimespec(ValuationDate, Maturity, 4)
```

```
TimeSpec =

    FinObj: 'BinTimeSpec'
  ValuationDate: 731398
    Maturity: 732859
    NumPeriods: 4
        Basis: 0
    EndMonthRule: 1
    tObs: [0 1 2 3 4]
    dObs: [1x5 double]
```

eqptimespec

See Also

eqptree, stockspec

Purpose Construct EQP stock tree

Syntax EQPTree = eqptree(StockSpec, RateSpec, TimeSpec)

Arguments

- StockSpec Stock specification. See stockspect for information on creating a stock specification.
- RateSpec Interest-rate specification for the initial risk free rate curve. See intenvset for information on declaring an interest-rate variable.
- TimeSpec Tree time layout specification. Defines the observation dates of the equal probabilities binomial tree. See eqptimespec for information on the tree structure.

Note The standard equal probabilities tree assumes a constant interest rate, but RateSpec allows you to specify an interest-rate curve with varying rates. If you specify variable interest rates, the resulting tree will not be a standard equal probabilities tree.

Description EQPTree = eqptree(StockSpec, RateSpec, TimeSpec) constructs an equal probabilities stock tree.

EQPTree is a MATLAB structure specifying the time layout for an equal probabilities stock tree.

Examples Using the data provided, create a stock specification (StockSpec), rate specification (RateSpec), and tree time layout specification (TimeSpec). Then use these specifications to create a CRR tree with crrtree.

```
Sigma = 0.20;
AssetPrice = 50;
DividendType = 'cash';
```

```
DividendAmounts = [0.50; 0.50; 0.50; 0.50];
ExDividendDates = {'03-Jan-2003'; '01-Apr-2003'; '05-July-2003';
'01-Oct-2003'}
```

```
StockSpec = stockspec(Sigma, AssetPrice, DividendType, ...
DividendAmounts, ExDividendDates)
StockSpec =
```

```
    FinObj: 'StockSpec'
    Sigma: 0.2000
    AssetPrice: 50
    DividendType: 'cash'
    DividendAmounts: [4x1 double]
    ExDividendDates: [4x1 double]
```

```
RateSpec = intenvset('Rates', 0.05, 'StartDates',...
'01-Jan-2003', 'EndDates', '31-Dec-2003')
```

```
RateSpec =
```

```
    FinObj: 'RateSpec'
    Compounding: 2
    Disc: 0.9519
    Rates: 0.0500
    EndTimes: 1.9945
    StartTimes: 0
    EndDates: 731946
    StartDates: 731582
    ValuationDate: 731582
    Basis: 0
    EndMonthRule: 1
```

```
ValuationDate = '1-Jan-2003';
Maturity = '31-Dec-2003';
TimeSpec = eqptimespec(ValuationDate, Maturity, 4)
```

```
TimeSpec =
```

```

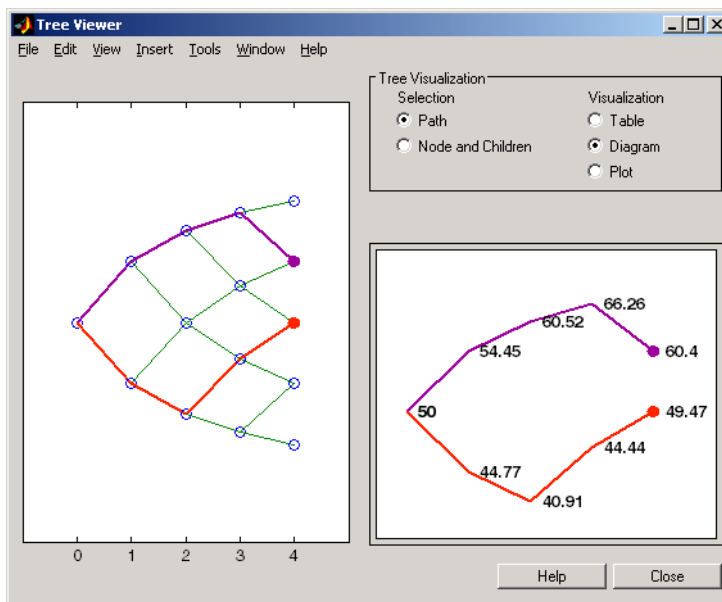
        FinObj: 'BinTimeSpec'
ValuationDate: 731582
        Maturity: 731946
        NumPeriods: 4
        Basis: 0
        EndMonthRule: 1
EQPTree = eqptree(StockSpec, RateSpec, TimeSpec)

EQPTree =

        FinObj: 'BinStockTree'
        Method: 'EQP'
StockSpec: [1x1 struct]
TimeSpec: [1x1 struct]
RateSpec: [1x1 struct]
        tObs: [0 0.2493 0.4986 0.7479 0.9972]
        dObs: [731582 731672 731763 731856 731946]
        STree: {1x5 cell}
        UpProbs: [0.5000 0.5000 0.5000 0.5000]

```

Use treeviewer to observe the tree you have created.



See Also eqptimespec, intenvset, stockspec

Purpose Price fixed-rate note from BDT interest-rate tree

Syntax [Price, PriceTree] = fixedbybdt(BDTree, CouponRate, Settle, Maturity, Reset, Basis, Principal, Options)

Arguments

BDTree	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) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none">• 0 = actual/actual (default)• 1 = 30/360 (SIA)• 2 = actual/360• 3 = actual/365• 4 = 30/360 (PSA)• 5 = 30/360 (ISDA)• 6 = 30/360 (European)• 7 = actual/365 (Japanese)• 8 = actual/actual (ISMA)• 9 = actual/360 (ISMA)

- 10 = actual/365 (ISMA)
- Principal (Optional) The notional principal amount. Default = 100.
- Options (Optional) Derivatives pricing options structure created with `derivset`.

Description

`[Price, PriceTree] = fixedbybdt(BDTree, 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.

Examples

Price a 10% fixed-rate note using a BDT interest-rate tree.

Load the file `deriv.mat`, which provides `BDTree`. The `BDTree` structure contains the time and interest-rate information needed to price the note.

```
load deriv.mat
```

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(BDTree, CouponRate, Settle, Maturity, Reset)
```

```
Price =
```

```
92.9974
```

See Also

`bdttree`, `bondbybdt`, `capbybdt`, `cfbybdt`, `floatbybdt`, `floorbybdt`, `swapbybdt`

Purpose Price fixed-rate note from Black-Karasinski interest-rate tree

Syntax [Price, PriceTree] = fixedbybk(BKTree, CouponRate, Settle, Maturity, Reset, Basis, Principal, Options)

Arguments

BKTree	Interest-rate tree structure created by bktree.
CouponRate	Decimal annual rate.
Settle	Settlement dates. 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) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none">• 0 = actual/actual (default)• 1 = 30/360 (SIA)• 2 = actual/360• 3 = actual/365• 4 = 30/360 (PSA)• 5 = 30/360 (ISDA)• 6 = 30/360 (European)• 7 = actual/365 (Japanese)• 8 = actual/actual (ISMA)• 9 = actual/360 (ISMA)

- 10 = actual/365 (ISMA)
- 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] = fixedbybk(BKTree, CouponRate, Settle, Maturity, Reset, Basis, Principal, Options)` computes the price of a fixed-rate note from a Black-Karasinski 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 BK tree. The fixed-rate note argument `Settle` is ignored.

Examples

Price a 5% fixed-rate note using a Black-Karasinski interest-rate tree.

Load the file `deriv.mat`, which provides `BKTree`. The `BKTree` structure contains the time and interest-rate information needed to price the note.

```
load deriv.mat;
```

Set the required values. Other arguments will use defaults.

```
CouponRate = 0.05;
Settle = '01-Jan-2005';
Maturity = '01-Jan-2006';
```

fixedbybk

Use `fixedbybk` to compute the price of the note.

```
Price = fixedbybk(BKTree, CouponRate, Settle, Maturity)
```

```
Price =
```

```
103.5126
```

See Also

`bktree`, `bondbybk`, `capbybk`, `cfbybk`, `floatbybk`, `floorbybk`, `swapbybk`

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) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none">• 0 = actual/actual (default)• 1 = 30/360 (SIA)• 2 = actual/360• 3 = actual/365• 4 = 30/360 (PSA)• 5 = 30/360 (ISDA)• 6 = 30/360 (European)• 7 = actual/365 (Japanese)• 8 = actual/actual (ISMA)• 9 = actual/360 (ISMA)

- 10 = actual/365 (ISMA)
- 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.

Examples

Price a 4% fixed-rate note using an HJM forward-rate tree.

Load the file `deriv.mat`, which provides `HJMTree`. The `HJMTree` structure contains the time and forward-rate information needed to price the note.

```
load deriv.mat
```

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 from Hull-White interest-rate tree

Syntax `[Price, PriceTree] = floatbybj(HWTree, CouponRate, Settle, Maturity, Reset, Basis, Principal, Options)`

Arguments

HWTree	Interest-rate tree structure created by hwtree.
CouponRate	Decimal annual rate.
Settle	Settlement dates. 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) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none">• 0 = actual/actual (default)• 1 = 30/360 (SIA)• 2 = actual/360• 3 = actual/365• 4 = 30/360 (PSA)• 5 = 30/360 (ISDA)• 6 = 30/360 (European)• 7 = actual/365 (Japanese)• 8 = actual/actual (ISMA)• 9 = actual/360 (ISMA)

- 10 = actual/365 (ISMA)
- 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] = floatbybj(HWTree, CouponRate, Settle, Maturity, Reset, Basis, Principal, Options)` computes the price of a fixed-rate note from a Hull-White 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 HW tree. The fixed-rate note argument `Settle` is ignored.

Examples

Price a 5% fixed-rate note using a Hull-White interest-rate tree.

Load the file `deriv.mat`, which provides `HWTree`. The `HWTree` structure contains the time and interest-rate information needed to price the note.

```
load deriv.mat;
```

Set the required values. Other arguments will use defaults.

```
CouponRate = 0.05;
Settle = '01-Jan-2005';
Maturity = '01-Jan-2006';
```

fixedbyhw

Use `fixedbyhw` to compute the price of the note.

```
Price = fixedbyhw(HWTree, CouponRate, Settle, Maturity)
```

```
Price =
```

```
103.5126
```

See Also

`bondbyhw`, `capbyhw`, `cfbyhw`, `floatbyhw`, `floorbyhw`, `hwtree`, `swapbyhw`

Purpose Price fixed-rate note from set of zero curves

Syntax Price = fixedbyzero(RateSpec, CouponRate, Settle, Maturity, Reset, Basis, Principal)

Arguments

RateSpec	Structure containing the properties of an interest-rate structure. See <code>intenvset</code> 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 of the instrument. A vector of integers. <ul style="list-style-type: none">• 0 = actual/actual (default)• 1 = 30/360 (SIA)• 2 = actual/360• 3 = actual/365• 4 = 30/360 (PSA)• 5 = 30/360 (ISDA)• 6 = 30/360 (European)• 7 = actual/365 (Japanese)• 8 = actual/actual (ISMA)• 9 = actual/360 (ISMA)

- 10 = actual/365 (ISMA)

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 from 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.mat
```

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`

floatbybdt

Purpose Price floating-rate note from BDT interest-rate tree

Syntax [Price, PriceTree] = floatbybdt(BDTree, Spread, Settle, Maturity, Reset, Basis, Principal, Options)

Arguments

BDTree	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) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none">• 0 = actual/actual (default)• 1 = 30/360 (SIA)• 2 = actual/360• 3 = actual/365• 4 = 30/360 (PSA)• 5 = 30/360 (ISDA)• 6 = 30/360 (European)• 7 = actual/365 (Japanese)• 8 = actual/actual (ISMA)• 9 = actual/360 (ISMA)

- 10 = actual/365 (ISMA)
- 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] = floatbybdt(BDTree, 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 `BDTree`. The `BDTree` structure contains the time and interest-rate information needed to price the note.

```
load deriv.mat;
```

Set the required values. Other arguments will use defaults.

```
Spread = 20;
Settle = '01-Jan-2000';
Maturity = '01-Jan-2003';
```

floatbybdt

Use `floatbybdt` to compute the price of the note.

```
Price = floatbybdt(BDTree, Spread, Settle, Maturity)
```

```
Price =
```

```
100.4865
```

See Also

`bdttree`, `bondbybdt`, `capbybdt`, `cfbybdt`, `fixedbybdt`, `floorbybdt`,
`swapbybdt`

Purpose Price floating-rate note from Black-Karasinski interest-rate tree

Syntax [Price, PriceTree] = floatbybk(BKTree, Spread, Settle, Maturity, Reset, Basis, Principal, Options)

Arguments

BKTree	Interest-rate tree structure created by bktree.
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) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none">• 0 = actual/actual (default)• 1 = 30/360 (SIA)• 2 = actual/360• 3 = actual/365• 4 = 30/360 (PSA)• 5 = 30/360 (ISDA)• 6 = 30/360 (European)• 7 = actual/365 (Japanese)• 8 = actual/actual (ISMA)• 9 = actual/360 (ISMA)

- 10 = actual/365 (ISMA)
- 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] = floatbybk(BKTree, Spread, Settle, Maturity, Reset, Basis, Principal, Options)` computes the price of a floating-rate note from a Black-Karasinski 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 BK tree. The floating-rate note argument `Settle` is ignored.

Examples

Price a 20 basis point floating-rate note using a Black-Karasinski interest-rate tree.

Load the file `deriv.mat`, which provides `BKTree`. The `BKTree` structure contains the time and interest-rate information needed to price the note.

```
load deriv.mat;
```

Set the required values. Other arguments will use defaults.

```
Spread = 20;  
Settle = '01-Jan-2005';  
Maturity = '01-Jan-2006';
```


Use floatbybk to compute the price of the note.

```
Price = floatbybk(BKTree, Spread, Settle, Maturity)
```

```
Price =
```

```
100.3825
```

See Also

bktree, bondbybk, capbybk, cfbybk, fixedbybk, floorbybk, swapbybk

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) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none">• 0 = actual/actual (default)• 1 = 30/360 (SIA)• 2 = actual/360• 3 = actual/365• 4 = 30/360 (PSA)• 5 = 30/360 (ISDA)• 6 = 30/360 (European)• 7 = actual/365 (Japanese)• 8 = actual/actual (ISMA)• 9 = actual/360 (ISMA)

- 10 = actual/365 (ISMA)
- 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`. The `HJMTree` structure contains the time and forward-rate information needed to price the note.

```
load deriv.mat
```

Set the required values. Other arguments will use defaults.

```
Spread = 20;
Settle = '01-Jan-2000';
Maturity = '01-Jan-2003';
```

floatbyhjm

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 from Hull-White interest-rate tree

Syntax

```
[Price, PriceTree] = floatbyhw(HWTree, Spread, Settle,  
Maturity, Reset, Basis, Principal, Options)
```

Arguments

HWTree	Interest-rate tree structure created by hwtree.
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) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none">• 0 = actual/actual (default)• 1 = 30/360 (SIA)• 2 = actual/360• 3 = actual/365• 4 = 30/360 (PSA)• 5 = 30/360 (ISDA)• 6 = 30/360 (European)• 7 = actual/365 (Japanese)• 8 = actual/actual (ISMA)• 9 = actual/360 (ISMA)

- 10 = actual/365 (ISMA)
- 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] = floatbyhw(HWTree, Spread, Settle, Maturity, Reset, Basis, Principal, Options)` computes the price of a floating-rate note from a Hull-White 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 HW tree. The floating-rate note argument `Settle` is ignored.

Examples

Price a 20 basis point floating-rate note using a Hull-White interest-rate tree.

Load the file `deriv.mat`, which provides `HWTree`. The `HWTree` structure contains the time and interest-rate information needed to price the note.

```
load deriv.mat
```

Set the required values. Other arguments will use defaults.

```
Spread = 20;  
Settle = '01-Jan-2005';  
Maturity = '01-Jan-2006';
```

Use `floatbyhw` to compute the price of the note.

```
Price = floatbyhw(HWTree, Spread, Settle, Maturity)
```

```
Price =
```

```
100.3825
```

See Also

`bondbyhw`, `capbyhw`, `cfbyhw`, `fixedbyhw`, `floorbyhw`, `hwtree`, `swapbyhw`

floatbyzero

Purpose Price floating-rate note from set of zero curves

Syntax Price = floatbyzero(RateSpec, Spread, Settle, Maturity, Reset, Basis, Principal)

Arguments

RateSpec	Structure containing the properties of an interest-rate structure. See <code>intenvset</code> 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 of the instrument. A vector of integers. <ul style="list-style-type: none">• 0 = actual/actual (default)• 1 = 30/360 (SIA)• 2 = actual/360• 3 = actual/365• 4 = 30/360 (PSA)• 5 = 30/360 (ISDA)• 6 = 30/360 (European)• 7 = actual/365 (Japanese)• 8 = actual/actual (ISMA)• 9 = actual/360 (ISMA)

- 10 = actual/365 (ISMA)

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 from 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.mat
```

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
```

floatbyzero

See Also

`bondbyzero`, `cfbyzero`, `fixedbyzero`, `swapbyzero`

Purpose Price floor instrument from BDT interest-rate tree

Syntax [Price, PriceTree] = floorbybdt(BDTree, Strike, Settle, Maturity, Reset, Basis, Principal, Options)

Arguments

BDTree	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) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none"> • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (PSA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese)

- 8 = actual/actual (ISMA)
- 9 = actual/360 (ISMA)
- 10 = actual/365 (ISMA)

Principal (Optional) The notional principal amount. Default = 100.

Options (Optional) Derivatives pricing options structure created with `derivset`.

Description

`[Price, PriceTree] = floorbybdt(BDTree, 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 `BDTree`. `BDTree` contains the time and interest-rate information needed to price the floor instrument.

```
load deriv.mat;
```

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(BDTree, 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. Remaining arguments will use defaults.

```
FloorStrike = 0.10;
Settlement = ValuationDate;
Maturity = '01-01-2002';
FloorReset = 1;
```

Finally, use floorbybdt to find the price of the floor instrument.

floorbybdt

```
Price= floorbybdt(BDTree, FloorStrike, Settlement, Maturity,...  
FloorReset)
```

```
Price =
```

```
0.0431
```

See Also

bdttree, capbybdt, cfbybdt, swapbybdt

Purpose Price floor instrument from Black-Karasinski interest-rate tree

Syntax [Price, PriceTree] = floorbybk(BKTree, Strike, Settle, Maturity, Reset, Basis, Principal, Options)

Arguments

HWTree	Interest-rate tree structure created by bktree.
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 BK 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) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none"> • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (PSA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese)

- 8 = actual/actual (ISMA)
- 9 = actual/360 (ISMA)
- 10 = actual/365 (ISMA)

Principal (Optional) The notional principal amount. Default = 100.

Options (Optional) Derivatives pricing options structure created with `derivset`.

Description

`[Price, PriceTree] = floorbybk(BKTree, Strike, Settlement, Maturity, Reset, Basis, Principal, Options)` computes the price of a floor instrument from a Black-Karasinski 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 a Black-Karasinski interest-rate tree.

Load the file `deriv.mat`, which provides `BKTree`. The `BKTree` structure contains the time and interest rate information needed to price the floor instrument.

```
load deriv.mat;
```

Set the required values. Other arguments will use defaults.

```
Strike = 0.03;  
Settle = '01-Jan-2005';  
Maturity = '01-Jan-2009';
```

Use `floorbyhw` to compute the price of the floor instrument.

```
Price = floorbybk(BKTree, Strike, Settle, Maturity)
```


Price =

0.2061

See Also

bktree, capbybk, cfbybk, swapbybk

floorbyhjm

Purpose Price floor instrument from 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) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none">• 0 = actual/actual (default)• 1 = 30/360 (SIA)• 2 = actual/360• 3 = actual/365• 4 = 30/360 (PSA)• 5 = 30/360 (ISDA)• 6 = 30/360 (European)• 7 = actual/365 (Japanese)

- 8 = actual/actual (ISMA)
 - 9 = actual/360 (ISMA)
 - 10 = actual/365 (ISMA)
- 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`. The `HJMTree` structure contains the time and forward-rate information needed to price the floor instrument.

```
load deriv.mat;
```

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)
```

floorbyhjm

Price =

0.0486

See Also

capbyhjm, cfbyhjm, hjmtree, swapbyhjm

Purpose Price floor instrument from Hull-White interest-rate tree

Syntax [Price, PriceTree] = floorbyhw(HWTree, Strike, Settle, Maturity, Reset, Basis, Principal, Options)

Arguments

HWTree	Interest-rate tree structure created by hwtree.
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 HW 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) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none"> • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (PSA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese)

- 8 = actual/actual (ISMA)
- 9 = actual/360 (ISMA)
- 10 = actual/365 (ISMA)

Principal (Optional) The notional principal amount. Default = 100.

Options (Optional) Derivatives pricing options structure created with `derivset`.

Description

`[Price, PriceTree] = floorbyhw(HWTree, Strike, Settlement, Maturity, Reset, Basis, Principal, Options)` computes the price of a floor instrument from an HW 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 a Hull-White interest-rate tree.

Load the file `deriv.mat`, which provides `HWTree`. The `HWTree` structure contains the time and interest rate information needed to price the floor instrument.

```
load deriv.mat;
```

Set the required values. Other arguments will use defaults.

```
Strike = 0.03;  
Settle = '01-Jan-2005';  
Maturity = '01-Jan-2009';
```

Use `floorbyhw` to compute the price of the floor instrument.

```
Price = floorbyhw(HWTree, Strike, Settle, Maturity)
```

Price =

0.4616

See Also

capbyhw, cfbyhw, hwtree, swapbyhw

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 $A * PortWts \leq b$, where $A = ConSet(:, 1:end-1)$ and $b = ConSet(:, end)$.

Notes

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.

`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.

Not all problems are solvable (for example, 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 NaNs 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 lets investors 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 Financial Toolbox documentation

`lsqlin` in the Optimization Toolbox documentation

hedgeslf

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 <code>FixedInd = 1</code> ; 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 <code>FixedInd = []</code> .
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, <code>PortHolds</code> , satisfies all the inequalities $A * PortHolds \leq b$, 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];
```

hedgeslf

```
Price = [1.2; 1.0];
W0 = [1; 1];
[PortSens, PortValue, PortHolds]= hedgeslf(Sens, Price, W0)

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];
W0 = [1; 1];
ConSet = pcalims([2 2])

% O.K. if nothing fixed.

[PortSens, PortValue, PortHolds]= hedgeslf(Sens, Price, W0,...
[], ConSet)

PortSens =

    2.8800
    0.6400

PortValue =
```

```

4.4000

PortHolds =

    2
    2

% W0(1) is not greater than 2.

[PortSens, PortValue, PortHolds] = hedgeslf(Sens, Price, W0,...
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];
W0 = [1; 1];
ConSet = pcalims([2 2],[1 1]);

[PortSens, PortValue, PortHolds] = hedgeslf(Sens, Price, W0,...
[],ConSet)

??? Error using ==> hedgeslf
Overly restrictive allocation constraints specified in ConSet

```

See Also

[hedgeopt](#)
[lsqlin](#) in the Optimization Toolbox documentation
[portcons](#) in Financial Toolbox documentation

hjmprice

Purpose Instrument prices from HJM interest-rate tree

Syntax `Price = hjmprice(HJMTree, InstSet, Options)`

Arguments

HJMTree	Heath-Jarrow-Morton tree sampling a forward-rate process. See <code>hjmtree</code> 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 <code>derivset</code> .

Description

`Price = hjmprice(HJMTree, InstSet, Options)` computes arbitrage free prices for instruments using an interest-rate tree created with `hjmtree`. A subset of 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 from an HJM tree.
- capbyhjm: Price a cap from an HJM tree.
- cfbyhjm: Price an arbitrary set of cash flows from an HJM tree.
- fixedbyhjm: Price a fixed-rate note from an HJM tree.
- floatbyhjm: Price a floating-rate note from an HJM tree.
- floorbyhjm: Price a floor from an HJM tree.
- optbndbyhjm: Price a bond option from an HJM tree.
- swapbyhjm: Price a swap from 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'
```

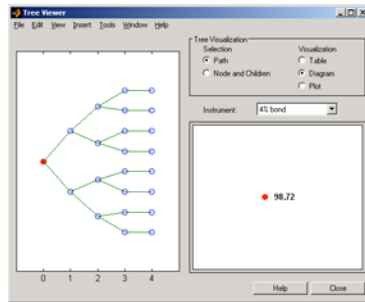
```
PBush: {1x5 cell}
```

```
AIBush: {1x5 cell}
```

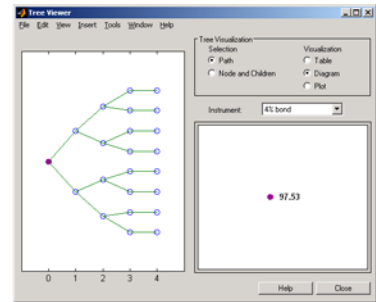
```
tObs: [0 1 2 3 4]
```

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

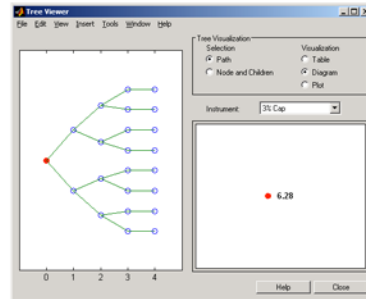
```
treeview(PriceTree, HJMSubset)
```



First 4% Bond (Maturity 2003)



Second 4% Bond (Maturity 2004)



3% Cap

See Also

hjmsens, hjmtree, hjmvolspec, instadd, intenvprice, intenvsens

Purpose Instrument prices and sensitivities from 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 <code>hjmtree</code> 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 <code>derivset</code> .

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

[hjprice](#), [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	<p>(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:</p> <p>Compounding = 1, 2, 3, 4, 6, 12</p> <p>Disc = $(1 + Z/F)^{-T}$, where F is the compounding frequency, Z is the zero rate, and T is the time in periodic units; for example, T = F is one year.</p> <p>Compounding = 365</p> <p>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.</p> <p>Compounding = -1</p> <p>Disc = $\exp(-T*Z)$, where T is time in years.</p>

hjmtimespec

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 Construct HJM interest-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 $\sigma(t, T)$ for each factor. See <code>hjmvolspec</code> for information on the volatility process.
RateSpec	Interest-rate specification for the initial rate curve. See <code>intenvset</code> 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 <code>hjmtimespec</code> 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 an HJM volatility specification (VolSpec), rate specification (RateSpec), and tree time layout specification (TimeSpec). Then use these specifications to create an HJM tree using `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];
```

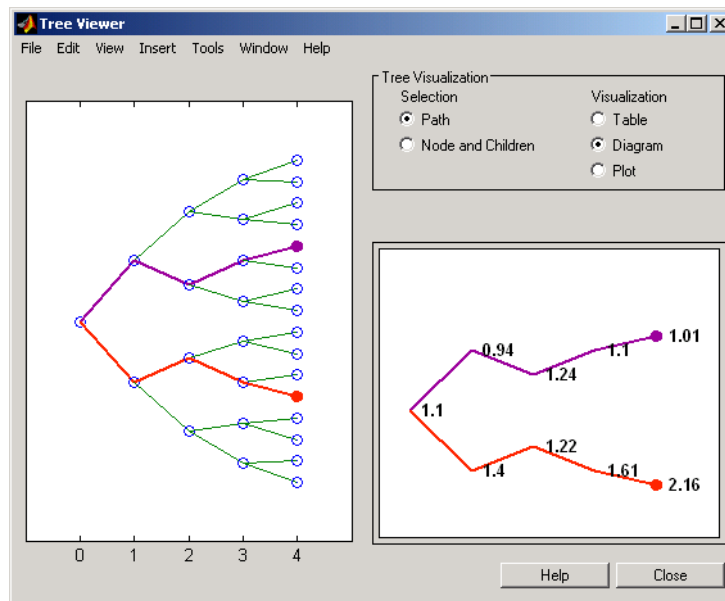
```
HJMVolSpec = hjmvolspec('Stationary', Volatility , CurveTerm);

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 HJM interest-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 following table 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.
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	$\sigma(t, T) = \text{Sigma_0}$
Stationary	$\sigma(t, T) = \text{Vol}(T-t) = \text{Vol}(\text{Term})$
Exponential	$\sigma(t, T) = \text{Sigma_0} * \exp(-\text{Lambda} * (T-t))$

Volatility Specification	Formula
Vasicek, Hull-White	$\sigma(t, T) = \text{Sigma}_0 * \exp(-\text{Decay}(T-t))$
Proportional	$\sigma(t, T) = \text{Prop}(T-t) * \max(\text{SpotRate}(t), \text{MaxSpot})$

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'
```

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](#)

Purpose Instrument prices from Hull-White interest-rate tree

Syntax [Price, PriceTree] = hwprice(HWTree, InstSet, Options)

Arguments

HWTree	Interest-rate tree structure created by hwtree.
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] = hwprice(HWTree, InstSet, Options) computes arbitrage free prices for instruments using an interest-rate tree created with hwtree. 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.

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

Related single-type pricing functions are

- `bondbyhw`: Price a bond from a Hull-White tree.
- `capbyhw`: Price a cap from a Hull-White tree.
- `cfbyhw`: Price an arbitrary set of cash flows from a Hull-White tree.
- `fixedbyhw`: Price a fixed-rate note from a Hull-White tree.
- `floatbyhw`: Price a floating-rate note from a Hull-White tree.
- `floorbyhw`: Price a floor from a Hull-White tree.
- `optbndbyhw`: Price a bond option from a Hull-White tree.
- `swapbyhw`: Price a swap from a Hull-White tree.

Examples

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

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

```
instdisp(HWSubSet)
```

Index	Type	CouponRate	Settle	Maturity	Period	Name ...
1	Bond	0.04	01-Jan-2004	01-Jan-2007	1	4% bond
2	Bond	0.04	01-Jan-2004	01-Jan-2008	1	4% bond

Index	Type	Strike	Settle	Maturity	CapReset...	Name ...
3	Cap	0.06	01-Jan-2004	01-Jan-2008	1	6% Cap

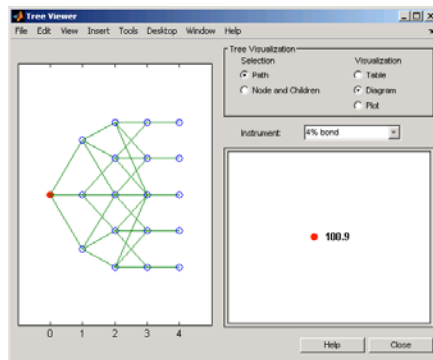
```
[Price, PriceTree] = hwprice(HWTree, HWSubSet);
```

```
Price =
```

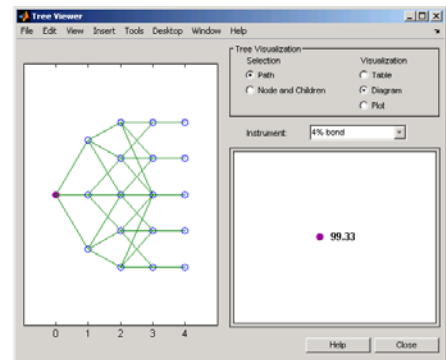
```
100.9188  
99.3296  
0.5837
```


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

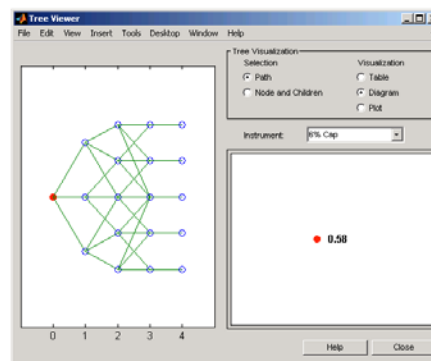
```
treeviewer(PriceTree, HWSubSet)
```



First 4% Bond (Maturity 2007)



Second 4% Bond (Maturity 2008)



6% Cap

See Also

hwsens, hwtree, instadd, intenvprice, intenvsens

Purpose Instrument prices and sensitivities from HW interest-rate tree

Syntax [Delta, Gamma, Vega, Price] = hwsens(HWTree, InstSet, Options)

Arguments

HWTree	Interest-rate tree structure created by hwtree.
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] = hwsens(HWTree, InstSet, Options) computes instrument sensitivities and prices for instruments using an interest-rate tree created with the hwtree function. NINST instruments from a financial instrument variable, InstSet, are priced. hwsens 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 hwtree. See hwtree 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 hwtree.

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 `hwtree`. See `hwvolspec` 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;
HWSubSet = instselect(HWInstSet, 'Type', {'Bond', 'Cap'});

instdisp(HWSubSet)
```

Index	Type	CouponRate	Settle	Maturity	Period	Name ...
1	Bond	0.04	01-Jan-2004	01-Jan-2007	1	4% Bond
2	Bond	0.04	01-Jan-2004	01-Jan-2008	1	4% Bond

Index	Type	Strike	Settle	Maturity	CapReset...	Name ...
3	Cap	0.06	01-Jan-2004	01-Jan-2008	1	6% Cap

```
[Delta, Gamma] = hwsens(HWTree, HWSubSet)

Delta =

    -291.26
    -374.64
     59.55
```

Gamma =

858.41

1460.88

4843.65

See Also

hwprice, hwtree, hwwolspec, instadd

Purpose Specify time structure for Hull-White tree

Syntax TimeSpec = hwtimespec(ValuationDate, Maturity, Compounding)

Arguments

ValuationDate Scalar date marking the pricing date and first observation in the tree. Specify as a 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 (continuous compounding). 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; for example, T = F is one year.

Compounding = 365

Disc = $(1 + Z/F)^{-T}$, where F is the number of days in the basis year and T is a number of days elapsed computed by basis.

Compounding = -1

Disc = $\exp(-T*Z)$, where T is time in years.

hwtimespec

Description

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

TimeSpec is a structure specifying the time layout for hwtree. 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 a four-period tree with annual nodes. Use annual compounding to report rates.

```
ValuationDate = 'Jan-1-2004';  
Maturity = ['12-31-2004'; '12-31-2005'; '12-31-2006';  
           '12-31-2007'];  
Compounding = 1;  
TimeSpec = hwtimespec(ValuationDate, Maturity, Compounding)
```

```
TimeSpec =  
  
           FinObj: 'HWTimeSpec'  
ValuationDate: 731947  
           Maturity: [4x1 double]  
           Compounding: 1  
           Basis: 0
```

See Also

bktree, hwtree, hwwolspec

Purpose Construct Hull-White interest-rate tree

Syntax HWTtree = hwtree(VolSpec, RateSpec, TimeSpec)

Arguments

VolSpec	Volatility process specification. See hwwolspec 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 HW tree and the compounding rule for date to time mapping and price-yield formulas. See hwtimespec for information on the tree structure.

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

Examples Using the data provided, create a Hull-White volatility specification (VolSpec), rate specification (RateSpec), and tree time layout specification (TimeSpec). Then use these specifications to create a Hull-White tree using hwtree.

```
Compounding = -1;
ValuationDate = '01-01-2004';
StartDate = ValuationDate;
VolDates = ['12-31-2004'; '12-31-2005'; '12-31-2006';
'12-31-2007'];
VolCurve = 0.01;
AlphaDates = '01-01-2008';
AlphaCurve = 0.1;
Rates = [0.0275; 0.0312; 0.0363; 0.0415];
```

```
HWVolSpec = hwwolspec(ValuationDate, VolDates, VolCurve,...
AlphaDates, AlphaCurve);

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

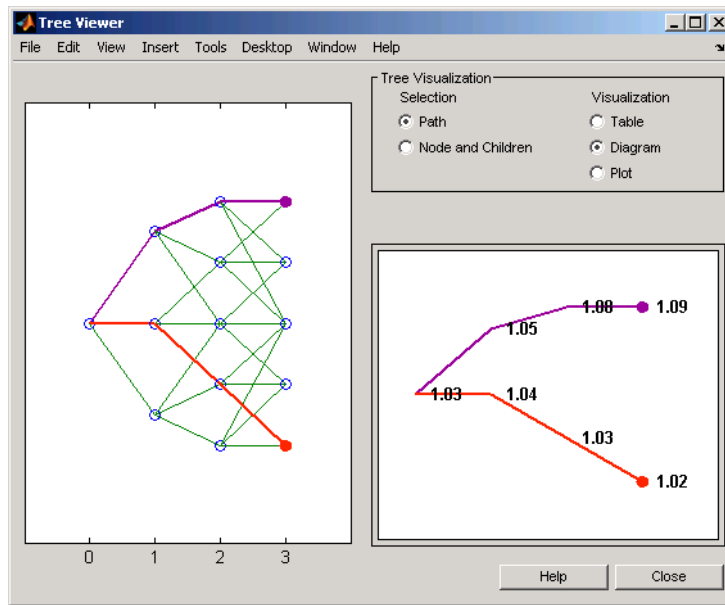
HWTimeSpec = hwtimespec(ValuationDate, VolDates, Compounding);
HWTTree = hwtree(HWVolSpec, RateSpec, HWTimeSpec)

HWTTree =

    FinObj: 'HWFwdTree'
    VolSpec: [1x1 struct]
    TimeSpec: [1x1 struct]
    RateSpec: [1x1 struct]
           tObs: [0 0.9973 1.9973 2.9973]
           dObs: [731947 732312 732677 733042]
    CFlowT: {[4x1 double] [3x1 double] [2x1 double] [3.9973]}
    Probs: {[3x1 double] [3x3 double] [3x5 double]}
    Connect: {[2] [2 3 4] [2 2 3 4 4]}
    FwdTree: {1x4 cell}
```

Use treeviewer to observe the tree you have created.

```
treeviewer(HWTTree)
```

See Also

hwprice, hwtimespec, hwwolspec, intenvset

hwvolspec

Purpose Specify Hull-White interest-rate volatility process

Syntax `VolSpec = hwvolspec(ValuationDate, VolDates, VolCurve, AlphaDates, AlphaCurve, InterpMethod)`

Arguments

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

Description `VolSpec = hwvolspec(ValuationDate, VolDates, VolCurve, AlphaDates, AlphaCurve, InterpMethod)` creates a structure specifying the volatility for `hwtree`.

Examples Using the data provided, create a Hull-White volatility specification (`VolSpec`).

```
ValuationDate = '01-01-2004';
StartDate = ValuationDate;
VolDates = ['12-31-2004'; '12-31-2005'; '12-31-2006';
'12-31-2007'];
VolCurve = 0.01;
AlphaDates = '01-01-2008';
```

```
AlphaCurve = 0.1;  
  
HWVolSpec = hwvolspec(ValuationDate, VolDates, VolCurve,...  
AlphaDates, AlphaCurve)  
  
HWVolSpec =  
  
    FinObj: 'HWVolSpec'  
ValuationDate: 731947  
    VolDates: [4x1 double]  
    VolCurve: [4x1 double]  
    AlphaCurve: 0.1000  
    AlphaDates: 733408  
VolInterpMethod: 'linear'
```

See Also [bktree](#), [interp1](#)

Purpose

Add types to instrument collection

Syntax

Arbitrary cash flow instrument. (See also instcf.)

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

Asian instrument. (See also instasian.)

```
InstSet = instasian('Asian', OptSpec, Strike, Settle, ExerciseDates, AmericanOpt, AvgType, AvgPrice, AvgDate)
```

Barrier instrument. (See also instbarrier.)

```
InstSet = instadd('Barrier', OptSpec, Strike, Settle, ExerciseDates, AmericanOpt, BarrierType, Barrier, Rebate)
```

Bond instrument. (See also instbond.)

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

Bond option. (See also instoptbnd.)

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

Cap instrument. (See also instcap.)

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

Compound instrument. (See also instcompound.)

```
InstSet = instadd('Compound', UOptSpec, UStrike, USettle, UExerciseDates, UAmericanOpt, COptSpec, CStrike, CSettle, CExerciseDates, CAmericanOpt)
```

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)
```

Floor instrument. (See also instfloor.)

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

Lookback instrument. (See also `instlookback`.)

```
InstSet = instadd('Lookback', OptSpec, Strike, Settle,  
ExerciseDates, AmericanOpt)
```

Stock option instrument. (See also `instoptstock`.)

```
InstSet = instadd('OptStock', OptSpec, Strike, Settle, Maturity,  
AmericanOpt)
```

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

<code>InstSetOld</code>	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.
-------------------------	---

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.

Description

`instadd` stores instruments of types 'Asian', 'Barrier', 'Bond', 'Cap', 'CashFlow', 'Compound', 'Fixed', 'Float', 'Floor', 'Lookback', 'OptBond', 'OptStock', or 'Swap'. This toolbox provides pricing and sensitivity routines for these instruments.

`InstSet` is an instrument set variable containing the new input data.

instadd

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

instasian, instbarrier, instbond, instcap, instcf, instcompound, instfixed, instfloat, instfloor, instlookback, instoptbnd, instoptstock, instswap

Purpose Add new instruments to instrument collection

Syntax

```
InstSet = instaddfield('FieldName', FieldList, '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 name 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 field names 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.

instaddfield

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)
```

```
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 = instadfield(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 = instadfield(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 = instadfield(InstSet,'Type','Option',...
'FieldName','Opt','Data','Put')
instdisp(InstSet)
```

instaddfield

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

Purpose Construct Asian option

Syntax `InstSet = instasian(InstSet, OptSpec, Strike, Settle, ExerciseDates, AmericanOpt, AvgType, AvgPrice, AvgDate)`
`[FieldList, ClassList, TypeString] = instasian`

Arguments

<code>InstSet</code>	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.
<code>OptSpec</code>	NINST-by-1 list of string values 'Call' or 'Put'.
<code>Strike</code>	NINST-by-1 vector of strike price values. Each row is the schedule for one option.
<code>Settle</code>	NINST-by-1 vector of Settle dates.
<code>ExerciseDates</code>	For a European option (<code>AmericanOpt = 0</code>): NINST-by-1 vector 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 (<code>AmericanOpt = 1</code>): NINST-by-2 vector of exercise date boundaries. For each instrument, the option can be exercised on any tree date between or including the pair of dates on that row. If only one non-NaN date is listed, or if <code>ExerciseDates</code> is NINST-by-1, the option can be exercised between the valuation date of the stock tree and the single listed exercise date.

AmericanOpt	(Optional) If AmericanOpt = 0, NaN, or is unspecified, the option is a European option. If AmericanOpt = 1, the option is an American option.
AvgType	(Optional) String = 'arithmetic' for arithmetic average (default) or 'geometric' for geometric average.
AvgPrice	(Optional) Scalar representing the average price of the underlying asset at Settle. This argument is used when AvgDate < Settle. Default is the current stock price.
AvgDate	(Optional) Scalar representing the date on which the averaging period begins. Default = Settle.

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 = instasian(InstSet, OptSpec, Strike, Settle, ExerciseDates, AmericanOpt, AvgType, AvgPrice, AvgDate)` specifies an Asian option.

`[FieldList, ClassList, TypeString] = instasian` 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 an Asian option instrument, `TypeString` = 'Asian'.

See Also instadd, instdisp, instget

instbarrier

Purpose Construct barrier option

Syntax `InstSet = instbarrier(InstSet, OptSpec, Strike, Settle, ExerciseDates, AmericanOpt, BarrierSpec, Barrier, Rebate)`
`[FieldList, ClassList, TypeString] = instbarrier`

Arguments

<code>InstSet</code>	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.
<code>OptSpec</code>	NINST-by-1 list of string values 'Call' or 'Put'.
<code>Strike</code>	NINST-by-1 vector of strike price values. Each row is the schedule for one option.
<code>Settle</code>	NINST-by-1 vector of Settle dates.
<code>ExerciseDates</code>	For a European option (<code>AmericanOpt = 0</code>): NINST-by-1 vector 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 (<code>AmericanOpt = 1</code>): NINST-by-2 vector of exercise date boundaries. For each instrument, the option can be exercised on any tree date between or including the pair of dates on that row. If only one non-NaN date is listed, or if <code>ExerciseDates</code> is NINST-by-1, the option can be exercised between the valuation date of the stock tree and the single listed exercise date.

AmericanOpt	If AmericanOpt = 0, NaN, or is unspecified, the option is a European option. If AmericanOpt = 1, the option is an American option.
BarrierSpec	List of string values: 'UI': Up Knock In 'UO': Up Knock Out 'DI': Down Knock In 'DO': Down Knock Out
Barrier	Vector of barrier values.
Rebate	(Optional) Vector of rebate values.

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 = instbarrier(InstSet, OptSpec, Strike, Settle, ExerciseDates, AmericanOpt, BarrierSpec, Barrier, Rebate) specifies a barrier option.

[FieldList, ClassList, TypeString] = instbarrier 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 barrier option instrument, TypeString = 'Barrier'.

See Also

instadd, instdisp, instget

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

<code>InstSet</code>	Instrument variable. This argument is specified only when adding bond instruments to an existing instrument set. See <code>instget</code> for more information on the <code>InstSet</code> variable.
<code>CouponRate</code>	Decimal number indicating the annual percentage rate used to determine the coupons payable on a bond.
<code>Settle</code>	Settlement date. A vector of serial date numbers or date strings. <code>Settle</code> must be earlier than or equal to <code>Maturity</code> .
<code>Maturity</code>	Maturity date. A vector of serial date numbers or date strings.
<code>Period</code>	(Optional) Coupons per year of the bond. A vector of integers. Allowed values are 1, 2, 3, 4, 6, and 12. Default = 2.

Basis	<p>(Optional) Day-count basis of the instrument. A vector of integers.</p> <ul style="list-style-type: none">• 0 = actual/actual (default)• 1 = 30/360 (SIA)• 2 = actual/360• 3 = actual/365• 4 = 30/360 (PSA)• 5 = 30/360 (ISDA)• 6 = 30/360 (European)• 7 = actual/365 (Japanese)• 8 = actual/actual (ISMA)• 9 = actual/360 (ISMA)• 10 = actual/365 (ISMA)
EndMonthRule	<p>(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.</p>
IssueDate	<p>(Optional) Date when a bond was issued.</p>
FirstCouponDate	<p>(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.</p>

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

Purpose Construct cap instrument

Syntax `InstSet = instcap(InstSet, Strike, Settle, Maturity, Reset, Basis, Principal)`
`[FieldList, ClassList, TypeString] = instcap`

Arguments

<code>InstSet</code>	Instrument variable. This argument is specified only when adding cap instruments to an existing instrument set. See <code>instget</code> for more information on the <code>InstSet</code> variable.
<code>Strike</code>	Rate at which the cap is exercised, as a decimal number.
<code>Settle</code>	Settlement date. Serial date number representing the settlement date of the cap.
<code>Maturity</code>	Serial date number representing the maturity date of the cap.
<code>Reset</code>	(Optional) NINST-by-1 vector representing the frequency of payments per year. Default = 1.
<code>Basis</code>	(Optional) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none">• 0 = actual/actual (default)• 1 = 30/360 (SIA)• 2 = actual/360• 3 = actual/365• 4 = 30/360 (PSA)• 5 = 30/360 (ISDA)• 6 = 30/360 (European)• 7 = actual/365 (Japanese)

- 8 = actual/actual (ISMA)
- 9 = actual/360 (ISMA)
- 10 = actual/365 (ISMA)

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

Purpose Construct cash flow instrument

Syntax `InstSet = instcf(InstSet, CFlowAmounts, CFlowDates, Settle, Basis)`
`[FieldList, ClassList, TypeString] = instcf`

Arguments

<code>InstSet</code>	Instrument variable. This argument is specified only when adding cash flow instruments to an existing instrument set. See <code>instget</code> for more information on the <code>InstSet</code> variable.
<code>CFlowAmounts</code>	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.
<code>CFlowDates</code>	NINST-by-MOSTCFS matrix of cash flow dates. Each entry contains the date of the corresponding cash flow in <code>CFlowAmounts</code> .
<code>Settle</code>	Settlement date on which the cash flows are priced.
<code>Basis</code>	(Optional) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none"> • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (PSA) • 5 = 30/360 (ISDA)

- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ISMA)
- 9 = actual/360 (ISMA)
- 10 = actual/365 (ISMA)

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; for example, `TypeString = 'CashFlow'`.

See Also

`instadd`, `instdisp`, `instget`, `intenvprice`

Purpose Construct compound option

Syntax `InstSet = instcompound(InstSet, UOptSpec, UStrike, USettle, UExerciseDates, UAmericanOpt, COptSpec, CStrike, CSettle, CExerciseDates, CAmericanOpt)`
`[FieldList, ClassList, TypeString] = instcompound`

Arguments

<code>InstSet</code>	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.
<code>UOptSpec</code>	String = 'Call' or 'Put'.
<code>UStrike</code>	1-by-1 vector of strike price values.
<code>USettle</code>	1-by-1 vector of Settle dates.
<code>UExerciseDates</code>	For a European option (<code>UAmericanOpt = 0</code>): 1-by-1 vector of exercise dates. For a European option, there is only one exercise date, the option expiry date. For an American option (<code>UAmericanOpt = 1</code>): 1-by-2 vector of exercise date boundaries. The option can be exercised on any tree date. If only one non-NaN date is listed, or if <code>ExerciseDates</code> is 1-by-1, the option can be exercised between the valuation date of the stock tree and the single listed exercise date.
<code>UAmericanOpt</code>	If <code>UAmericanOpt = 0</code> , NaN, or is unspecified, the option is a European option. If <code>UAmericanOpt = 1</code> , the option is an American option.

instcompound

COptSpec	NINST-by-1 list of string values 'Call' or 'Put' of the compound option.
CStrike	NINST-by-1 vector of strike price values. Each row is the schedule for one option.
CSettle	1-by-1 vector containing the settlement or trade date.
CExerciseDates	For a European option (CAmericanOpt = 0): NINST-by-1 vector 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 (CAmericanOpt = 1): NINST-by-2 vector of exercise date boundaries. For each instrument, the option can be exercised on any tree 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 valuation date of the stock tree and the single listed exercise date.
CAmericanOpt	If CAmericanOpt = 0, NaN, or is unspecified, the option is a European option. If CAmericanOpt = 1, the option is an American option.

Description

InstSet = instcompound(InstSet, UOptSpec, UStrike, USettle, UExerciseDates, UAmericanOpt, COptSpec, CStrike, CSettle, CExerciseDates, CAmericanOpt) specifies a compound option.

[FieldList, ClassList, TypeString] = instcompound 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 compound option instrument, `TypeString` = 'Compound'.

See Also

`instadd`, `instdisp`, `instget`

instdelete

Purpose Complement of instrument set by matching conditions

Syntax `ISubSet = instdelete(InstSet, 'FieldName', FieldList, 'Data', DataList, 'Index', IndexSet, 'Type', TypeList)`

Arguments

<code>InstSet</code>	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.
<code>FieldList</code>	String or number of fields (NFIELDS)-by-1 cell array of strings listing the name of each data field to match with data values.
<code>DataList</code>	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 <code>FieldList</code> . The number of columns is arbitrary and matching will ignore trailing NaNs or spaces.
<code>IndexSet</code>	(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.
<code>TypeList</code>	(Optional) String or number of types (NTYPES)-by-1 cell array of strings restricting instruments to match one of <code>TypeList</code> types. The default is all types in the instrument variable.

Note 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.

instdelete

```
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

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

instdisp

```
6      Option 95      2.9 Put      0
      Index Type Price Maturity      Contracts
7      TBill 99      01-Jul-1999  6
```

See Also

`datestr` in Financial Toolbox documentation

`num2str` in the MATLAB Reference

`instaddfield`, `instget`

Purpose List field names

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)` retrieves the 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

instfields

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 =
```

```
    'db1e'  
    'db1e'  
    'char'  
    'db1e'
```

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)
FieldList =
    'Delivery'
    'F'
    'Strike'
    'Opt'
    'Price'
    'Maturity'
    'Contracts'
```

See Also `instdisp`, `instlength`, `insttypes`

instfind

Purpose Search instruments for matching conditions

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

Arguments

<code>InstSet</code>	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.
<code>FieldList</code>	String or number of fields (NFIELDS)-by-1 cell array of strings listing the name of each data field to match with data values.
<code>DataList</code>	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 <code>FieldList</code> . The number of columns is arbitrary, and matching will ignore trailing NaNs or spaces.
<code>IndexSet</code>	(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.
<code>TypeList</code>	(Optional) String or number of types (NTYPES)-by-1 cell array of strings restricting instruments to match one of <code>TypeList</code> 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 =
```

instfind

```
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

Purpose Construct fixed-rate instrument

Syntax `InstSet = instfixed(InstSet, Strike, Settle, Maturity, Reset, Basis, Principal)`
`[FieldList, ClassList, TypeString] = instfixed`

Arguments

<code>InstSet</code>	Instrument variable. This argument is specified only when adding fixed-rate note instruments to an existing instrument set. See <code>instget</code> for more information on the <code>InstSet</code> variable.
<code>CouponRate</code>	Decimal annual rate.
<code>Settle</code>	Settlement date. Date string or serial date number representing the settlement date of the fixed-rate note.
<code>Maturity</code>	Date string or serial date number representing the maturity date of the fixed-rate note.
<code>Reset</code>	(Optional) NINST-by-1 vector representing the frequency of payments per year. Default = 1.
<code>Basis</code>	(Optional) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none">• 0 = actual/actual (default)• 1 = 30/360 (SIA)• 2 = actual/360• 3 = actual/365• 4 = 30/360 (PSA)• 5 = 30/360 (ISDA)• 6 = 30/360 (European)• 7 = actual/365 (Japanese)

- 8 = actual/actual (ISMA)
- 9 = actual/360 (ISMA)
- 10 = actual/365 (ISMA)

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`

Purpose Construct floating-rate instrument

Syntax `InstSet = instfloat(InstSet, Spread, Settle, Maturity, Reset, Basis, Principal)`
`[FieldList, ClassList, TypeString] = instfloat`

Arguments

<code>InstSet</code>	Instrument variable. This argument is specified only when adding floating-rate note instruments to an existing instrument set. See <code>instget</code> for more information on the <code>InstSet</code> variable.
<code>Spread</code>	Number of basis points over the reference rate.
<code>Settle</code>	Settlement date. Date string or serial date number representing the settlement date of the floating-rate note.
<code>Maturity</code>	Date string or serial date number representing the maturity date of the floating-rate note.
<code>Reset</code>	(Optional) NINST-by-1 vector representing the frequency of payments per year. Default = 1.
<code>Basis</code>	(Optional) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none">• 0 = actual/actual (default)• 1 = 30/360 (SIA)• 2 = actual/360• 3 = actual/365• 4 = 30/360 (PSA)• 5 = 30/360 (ISDA)• 6 = 30/360 (European)• 7 = actual/365 (Japanese)

instfloat

- 8 = actual/actual (ISMA)
- 9 = actual/360 (ISMA)
- 10 = actual/365 (ISMA)

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`

Purpose Construct floor instrument

Syntax `InstSet = instfloor(InstSet, Strike, Settle, Maturity, Reset, Basis, Principal)`
`[FieldList, ClassList, TypeString] = instfloor`

Arguments

<code>InstSet</code>	Instrument variable. This argument is specified only when adding floor instruments to an existing instrument set. See <code>instget</code> for more information on the <code>InstSet</code> variable.
<code>Strike</code>	Rate at which the floor is exercised, as a decimal number.
<code>Settle</code>	Settlement date. A vector of serial date numbers or date strings. <code>Settle</code> must be earlier than or equal to <code>Maturity</code> .
<code>Maturity</code>	Maturity date. A vector of serial date numbers or date strings.
<code>Reset</code>	(Optional) NINST-by-1 vector representing the frequency of payments per year. Default = 1.
<code>Basis</code>	(Optional) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none">• 0 = actual/actual (default)• 1 = 30/360 (SIA)• 2 = actual/360• 3 = actual/365• 4 = 30/360 (PSA)• 5 = 30/360 (ISDA)• 6 = 30/360 (European)

- 7 = actual/365 (Japanese)
- 8 = actual/actual (ISMA)
- 9 = actual/360 (ISMA)
- 10 = actual/365 (ISMA)

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`

Purpose Data from instrument variable

Syntax `[Data_1, Data_2,...,Data_n] = instget(InstSet, 'FieldName', FieldList, 'Index', IndexSet, 'Type', TypeList)`

Arguments

<code>InstSet</code>	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.
<code>FieldList</code>	(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. <code>FieldList</code> 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.
<code>IndexSet</code>	(Optional) Number of instruments (NINST)-by-1 vector of positions of instruments to work on. If <code>TypeList</code> is also entered, instruments referenced must be one of <code>TypeList</code> types and contained in <code>IndexSet</code> . The default is all indices available in the instrument variable.
<code>TypeList</code>	(Optional) String or number of types (NTYPES)-by-1 cell array of strings restricting instruments to match one of <code>TypeList</code> 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.

Description

[Data_1, Data_2,...,Data_n] = instget(InstSet, 'FieldName', FieldList, 'Index', IndexSet, 'Type', TypeList) retrieves 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 that 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,0] = instget(ExampleInst, 'Index', Ind, 'FieldName',...  
{ 'Type', 'Opt' })
```

```
T =
```

```
Option  
Futures  
Option  
TBill
```

```
0 =
```

```
Call
```

```
Put
```

Create a string report of holdings Type, Opt, and Value.

```
rstring = [T, 0, num2str(V(Ind))]
```

```
rstring =
```

```
Option Call    6800  
Futures       NaN  
Option Put    -7400  
TBill         594
```

See Also

instaddfield, instdisp, instgetcell

instgetcell

Purpose Data and context from instrument variable

Syntax `[DataList, FieldList, ClassList] =
instgetcell(InstSet, 'FieldName', FieldList, 'Index',
IndexSet, 'Type', TypeList)`

Arguments

<code>InstSet</code>	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.
<code>FieldList</code>	(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. <code>FieldList</code> should not be either <code>Type</code> or <code>Index</code> ; these field names are reserved. The default is all fields available for the returned set of instruments.
<code>IndexSet</code>	(Optional) Number of instruments (NINST)-by-1 vector of positions of instruments to work on. If <code>TypeList</code> is also entered, instruments referenced must be one of <code>TypeList</code> types and contained in <code>IndexSet</code> . The default is all indices available in the instrument variable.
<code>TypeList</code>	(Optional) String or number of types (NTYPES)-by-1 cell array of strings restricting instruments to match one of <code>TypeList</code> 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.

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
```

instgetcell

```
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`

instlookback

Purpose Construct lookback option

Syntax `InstSet = instlookback(InstSet, OptSpec, Strike, Settle, ExerciseDates, AmericanOpt)`
`[FieldList, ClassList, TypeString] = instlookback`

Arguments

<code>InstSet</code>	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.
<code>OptSpec</code>	NINST-by-1 list of string values 'Call' or 'Put'.
<code>Strike</code>	NINST-by-1 vector of strike price values. Each row is the schedule for one option.
<code>Settle</code>	NINST-by-1 vector of Settle dates.
<code>ExerciseDates</code>	For a European option (<code>AmericanOpt = 0</code>): NINST-by-1 vector 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 (<code>AmericanOpt = 1</code>): NINST-by-2 vector of exercise date boundaries. For each instrument, the option can be exercised on any tree date between or including the pair of dates on that row. If only one non-NaN date is listed, or if <code>ExerciseDates</code> is NINST-by-1, the option can be exercised between the valuation date of the stock tree and the single listed exercise date.
<code>AmericanOpt</code>	(Optional) If <code>AmericanOpt = 0</code> , NaN, or is unspecified, the option is a European option. If

AmericanOpt = 1, the option is an American option.

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 = instlookback(InstSet, OptSpec, Strike, Settle, ExerciseDates, AmericanOpt) specifies a lookback option.

[FieldList, ClassList, TypeString] = instlookback 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 lookback option instrument, TypeString = 'Lookback'.

See Also

instadd, instdisp, instget

instoptbnd

Purpose Construct bond option

Syntax

```
InstSet = instoptbnd(InstSet, BondIndex, OptSpec, Strike, ExerciseDates)
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'.

Note 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	<p>European option: NINST-by-1 vector of strike price values.</p> <p>Bermuda option: NINST by number of strikes (NSTRIKES) matrix of strike price values.</p> <p>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.</p> <p>For an American option:</p> <p>NINST-by-1 vector of strike price values for each option.</p>
ExerciseDates	<p>NINST-by-1 (European option) or NINST-by-NSTRIKES (Bermuda option) 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.</p> <p>For an American option:</p> <p>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 <code>Settle</code> and the single listed exercise date.</p>

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.

instoptbnd

`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, ClassList, TypeString] = instoptbnd` 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 option instrument, `TypeString = 'OptBond'`.

See Also

`hjmprice`, `instadd`, `instdisp`, `instget`

Purpose

Construct stock option

Syntax

```
InstSet = instoptstock(InstSet, OptSpec, Strike,  
ExerciseDates)  
InstSet = instoptstock(InstSet, OptSpec, Strike,  
ExerciseDates, AmericanOpt)  
[FieldList, ClassList, TypeString] = instoptstock
```

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. This argument is specified only when adding stock instruments to an existing instrument set. See <code>instget</code> for more information on the <code>InstSet</code> variable.
OptSpec	NINST-by-1 list of string values 'Call' or 'Put'.

Note 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	<p>European option: NINST-by-1 vector of strike price values.</p> <p>Bermuda option: NINST by number of strikes (NSTRIKES) matrix of strike price values.</p> <p>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.</p> <p>American option: NINST-by-1 vector of strike price values for each option.</p>
Settle	NINST-by-1 vector of settlement dates.
ExerciseDates	<p>NINST-by-1 (European option) or NINST-by-NSTRIKES (Bermuda option) 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.</p> <p>For an American option:</p> <p>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.</p>

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 = instoptstock(InstSet, OptSpec, Strike, ExerciseDates)` specifies a European or Bermuda option.

`InstSet = instoptstock(InstSet, 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, ClassList, TypeString] = instoptstock` 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 stock option instrument, `TypeString` = 'OptStock'.

See Also

`instadd`, `instdisp`, `instget`

instselect

Purpose Create instrument subset by matching conditions

Syntax `InstSubSet = instselect(InstSet, 'FieldName', FieldList, 'Data', DataList, 'Index', IndexSet, 'Type', TypeList)`

Arguments

<code>InstSet</code>	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.
<code>FieldList</code>	String or number of fields (NFIELDS)-by-1 cell array of strings listing the name of each data field to match with data values.
<code>DataList</code>	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 <code>FieldList</code> . The number of columns is arbitrary and matching will ignore trailing NaNs or spaces.
<code>IndexSet</code>	(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.
<code>TypeList</code>	(Optional) String or number of types (NTYPES)-by-1 cell array of strings restricting instruments to match one of <code>TypeList</code> 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. `'Index'` and `'Type'` arguments 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')
```

instselect

```
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	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.

instsetfield

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.
```

```
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

```

instsetfield

```
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, InstSetLegReset, Basis, Principal, LegType)
[FieldList, ClassList, TypeString] = instswap`

Arguments

<code>InstSet</code>	Instrument variable. This argument is specified only when adding a swap to an existing instrument set. See <code>instget</code> for more information on the <code>InstSet</code> variable.
<code>LegRate</code>	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.
<code>Settle</code>	Settlement date. NINST-by-1 vector of serial date numbers or date strings. <code>Settle</code> must be earlier than or equal to <code>Maturity</code> .
<code>Maturity</code>	Maturity date. NINST-by-1 vector of dates representing the maturity date for each swap.
<code>LegReset</code>	(Optional) NINST-by-2 matrix representing the reset frequency per year for each swap. Default = [1 1].

Basis	(Optional) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none">• 0 = actual/actual (default)• 1 = 30/360 (SIA)• 2 = actual/360• 3 = actual/365• 4 = 30/360 (PSA)• 5 = 30/360 (ISDA)• 6 = 30/360 (European)• 7 = actual/365 (Japanese)• 8 = actual/actual (ISMA)• 9 = actual/360 (ISMA)• 10 = actual/365 (ISMA)
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.

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, InstSetLegReset, 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

insttypes

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

List all of the types included in ExampleInst.

```
TypeList = insttypes(ExampleInst)
TypeList =
    'Futures'
    'Option'
    'TBill'
```

See Also

`instdisp`, `instfields`, `instlength`

intenvget

Purpose Properties of interest-rate structure

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

Arguments

<code>RateSpec</code>	A structure containing the properties of an interest-rate structure. See <code>intenvset</code> for information on creating <code>RateSpec</code> .
<code>ParameterName</code>	String indicating the parameter name to be accessed. The value of the named parameter is extracted from the structure <code>RateSpec</code> . 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`

intenvprice

Purpose Price instruments from set of zero curves

Syntax Price = intenvprice(RateSpec, InstSet)

Arguments

RateSpec	A structure containing 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 from a set of zero curves.
cfbyzero	Price arbitrary cash flow instrument from a set of zero curves.
fixedbyzero	Fixed-rate note prices from a set of zero curves.
floatbyzero	Floating-rate note prices from a set of zero curves.
swapbyzero	Swap prices from 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
```

```
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 from set of zero curves

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

Arguments

RateSpec	A structure containing the properties of an interest-rate structure. See <code>intenvset</code> 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 that use a zero coupon bond rate 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 ...	
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
3	Fixed	0.04	01-Jan-2000	01-Jan-2003	1	NaN	NaN	4% Fixed 80

```
[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
```

intenvsens

0.0033
1.0596

See Also

hmprice, hjmsens, instadd, intenvprice, intenvset

Purpose Set properties of interest-rate structure

Syntax [RateSpec, RateSpecOld] = intenvset(RateSpec, 'Argument1', Value1, 'Argument2', 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.

Arguments may be chosen from the following table 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; for example, T = F is one year.

Compounding = 365

$Disc = (1 + Z/F)^{-T}$, where F is the number of days in the basis year and T is a number of days elapsed computed by basis.

Compounding = -1

	$\text{Disc} = \exp(-T*Z)$, 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 instrument. A vector of integers. <ul style="list-style-type: none">• 0 = actual/actual (default)• 1 = 30/360 (SIA)• 2 = actual/360• 3 = actual/365• 4 = 30/360 (PSA)• 5 = 30/360 (ISDA)

- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ISMA)
- 9 = actual/360 (ISMA)
- 10 = actual/365 (ISMA)

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 argument names.

When creating a new `RateSpec`, the set of arguments 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 argument names and possible values.

Description

`[RateSpec, RateSpecOld] = intenvset(RateSpec, 'Argument1', Value1, 'Argument2', Value2, ...)` creates an interest term structure (`RateSpec`) in which the input argument list is specified as argument name /argument value pairs. The argument name portion of the pair must be recognized as a valid field of the output structure `RateSpec`; the argument value portion of the pair is then assigned to its paired field.

intenvset

If the optional argument `RateSpec` is specified, `intenvset` modifies an existing interest term structure `RateSpec` by changing the named argument to the specified values and recalculating the arguments 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 argument names and possible values.

`RateSpecOld` is a structure containing 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` argument 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 argument 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`

isafin

Purpose Returns True if input argument is financial structure type or financial object class

Syntax `IsFinObj = isafin(Obj, ClassName)`

Arguments

<code>Obj</code>	Name of a financial structure.
<code>ClassName</code>	String containing the name of a financial structure class.

Description `IsFinObj = isafin(Obj, ClassName)` returns True if input argument is a financial structure type or financial object class, otherwise False is returned.

Examples

```
load deriv.mat
IsFinObj = isafin(HJMTree, 'HJM_fwdTree') returns True
```

See Also `classfin`

Purpose Price instruments using an implied trinomial tree (ITT)

Syntax

```
Price = ittprice(ITTTree, InstSet)
Price = ittprice(ITTTree, InstSet, Options)
[Price, PriceTree] = ittprice(ITTTree, InstSet, Options)
```

Arguments

ITTTree	Implied trinomial stock tree. See <code>itttree</code> for information on creating the variable <code>ITTTree</code> .
InstSet	Variable containing a collection of NINST instruments. Instruments are broken down by type and each type can have different data fields.
Options	(Optional) Structure created using <code>derivset</code> containing derivative pricing options.

Description

```
Price = ittprice(ITTTree, InstSet)
Price = ittprice(ITTTree, InstSet, Options)
[Price, PriceTree] = ittprice(ITTTree, InstSet, Options)
```

The outputs for `ittprice` are:

- `Price` is a NINST-by-1 vector of prices of each instrument at time 0. The prices are computed by backward dynamic programming on the stock tree. If an instrument cannot be priced, a NaN is returned in that entry.
- `PriceTree` is a structure containing trees of vectors of instrument prices and a vector of observation times for each node.
 - `PriceTree.PTree` contains the prices.
 - `PriceTree.tObs` contains the observation times.
 - `PriceTree.dObs` contains the observation dates.

`ittprice` computes prices for instruments using an implied trinomial tree created with `itttree`.

Note `ittprice` handles the following instrument types: `optstock`, `barrier`, `Asian`, `lookback`, and `compound`. Use `instadd` to construct the defined types.

When using an implied trinomial tree, pricing of path-dependent options is done using Hull-White. Consequently, for these options there are no unique prices on the tree nodes with the exception of the root node. The corresponding nodes of the tree are populated with NaNs for these particular options. For information on single-type pricing functions to retrieve state-by-state pricing tree information, see the following:

- `barrierbyitt` for pricing barrier options using an ITT tree
- `optstockbyitt` for pricing American, European or Bermuda options using an ITT tree
- `asianbyitt` for pricing Asian options using an ITT tree
- `lookbackbyitt` for pricing lookback options using an ITT tree
- `compoundbyitt` for price compound options using an ITT tree

Examples

Load the ITT tree and instruments from the data file `deriv.mat`.

```
load deriv.mat
```

Price the barrier and Asian options contained in the instrument set.

```
ITTSubSet = instselect(ITTInstSet, 'Type', {'Barrier', 'Asian'});  
  
instdisp(ITTSubSet)
```



```

instdisp(ITTSubSet)
Index Type OptSpec Strike Settle ExerciseDates AmericanOpt BarrSpec Barr Rebate Name Quantity
1 Barrier call 85 01-Jan-2006 31-Dec-2008 1 ui 115 0 Barrier1 1

```

```

Index Type OptSpec Strike Settle ExerciseDates AmerOpt AvgType AvgPrice AvgDate Name Quantity
2 Asian call 55 01-Jan-2006 01-Jan-2008 0 arithmetic NaN NaN Asian1 5
3 Asian call 55 01-Jan-2006 01-Jan-2010 0 arithmetic NaN NaN Asian2 7

```

```
[Price, PriceTree] = ittprice(ITTree, ITTSubSet)
```

```
Price =
```

```
2.4074
```

```
3.2052
```

```
6.6074
```

```
PriceTree =
```

```
FinObj: 'TrinPriceTree'
```

```
PTree: {[3x1 double] [3x3 double] [3x5 double] [3x7 double] [3x9 double]}
```

```
tObs: [0 1 2 3 4]
```

```
dObs: [732678 733043 733408 733773 734139]
```

See Also

ittsens, itttree

Purpose Instrument sensitivities and prices using an implied trinomial tree (ITT)

Syntax

```
[Delta, Gamma, Vega] = ittsens(ITTree, InstSet)
[Delta, Gamma, Vega, Price] = ittsens(ITTree, InstSet)
[Delta, Gamma, Vega, Price] = ittsens(ITTree, InstSet,
Options)
```

Arguments

ITTree	Implied trinomial stock tree. See <code>itttree</code> for information on creating the variable <code>ITTree</code> .
InstSet	Variable containing a collection of NINST instruments. Instruments are broken down by type and each type can have different data fields.
Options	(Optional) Structure created using <code>derivset</code> containing derivative pricing options.

Description

```
[Delta, Gamma, Vega] = ittsens(ITTree, InstSet)
[Delta, Gamma, Vega, Price] = ittsens(ITTree, InstSet)
[Delta, Gamma, Vega, Price] = ittsens(ITTree, InstSet,
Options)
```

The outputs for `ittsens` are:

- Delta is a NINST-by-1 vector of deltas, representing the rate of change of instruments prices with respect to changes in the stock price.
- Gamma is a NINST-by-1 vector of gammas, representing the rate of change of instruments deltas with respect to changes in the stock price.
- Vega is a NINST-by-1 vector of vegas, representing the rate of change of instruments prices with respect to changes in the volatility of the stock. Vega is computed by finite differences in calls to `itttree`.

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

ittsens computes dollar sensitivities and prices for instruments using an ITT tree created with itttree.

Note ittsens handles the following instrument types: optstock, barrier, Asian, lookback, and compound. Use instadd to construct the defined types.

For path-dependent options (lookbacks and Asians), Delta and Gamma are computed by finite differences in calls to ittprice. For the rest of the options (optstock, barrier, and compound), Delta and Gamma are computed from the ITT tree and the corresponding option price tree.

All sensitivities are returned as dollar sensitivities. To find the per-dollar sensitivities, they must be divided by their respective instrument price.

Examples

Load the ITT tree and instruments from the data file deriv.mat. Compute the Delta and Gamma sensitivities of vanilla options and barrier option contained in the instrument set.

```
load deriv.mat
ITTSubSet = instselect(ITTInstSet, 'Type', {'OptStock', 'Barrier'});

instdisp(ITTSubSet)
>> instdisp(ITTSubSet)
```

Index	Type	OptSpec	Strike	Settle	ExerciseDates	AmericanOpt	Name	Quantity
1	OptStock	call	95	01-Jan-2006	31-Dec-2008	1	Call1	10
2	OptStock	put	80	01-Jan-2006	01-Jan-2010	0	Put1	4

```
Index Type OptSpec Strike Settle ExercDates AmerOpt BarrSpec Barr Rebate Name Quantity
3 Barrier call 85 01-Jan-2006 31-Dec-2008 1 ui 115 0 Barrier1 1
```

```
[Delta, Gamma] = ittsens(ITTree, ITSubSet)
```

```
Warning: The option price range specified in the input was too narrow.
Extrapolation was necessary, which may lead into unreliable results.
```

```
> In itttree>InterpOptPrices at 615
```

```
In itttree at 273
```

```
In stocktreesens>stocktreevega at 191
```

```
In stocktreesens at 92
```

```
In ittsens at 81
```

```
Delta =
```

```
0.2387
```

```
-0.4283
```

```
0.3482
```

References

Chriss, Neil. and I. Kawaller, *Black-Scholes and Beyond: Options Pricing Models*, McGraw-Hill, 1996, pp. 308-312.

See Also

ittprice, itttree

Purpose Specify time structure using an implied trinomial tree (ITT)

Syntax `TimeSpec = itttimespec(ValuationDate, Maturity, NumPeriods)`

Arguments

<code>ValuationDate</code>	Scalar date marking the pricing date and first observation in the tree. Specify <code>ValuationDate</code> as a serial date number or date string.
<code>Maturity</code>	Scalar date marking the depth of the tree.
<code>NumPeriods</code>	Scalar that determines how many time steps are in the tree.

Description `TimeSpec = itttimespec(ValuationDate, Maturity, NumPeriods)` creates the structure specifying the time layout for an ITT tree.

Examples Specify a 4-period tree with time steps of 1 year.

```
ValuationDate = '1-July-2006';  
Maturity = '1-July-2010';  
TimeSpec = itttimespec(ValuationDate, Maturity, 4);
```

See Also `itttree`, `stockspec`

Purpose Build an implied trinomial stock tree

Syntax `ittree(StockSpec, RateSpec, TimeSpec, StockOptSpec)`

Arguments

StockSpec	Stock specification. For more information, see <code>stockspec</code> .
RateSpec	Interest rate specification of the initial risk-free rate curve. For more information on declaring an interest rate variable, see <code>intenvset</code> .
TimeSpec	Tree time layout specification. Defines the observation dates of the implied trinomial tree. For more information on the tree structure, see <code>itttimespec</code> .
StockOptSpec	Option stock specification. For more information, see <code>stockoptspec</code> .

Description `ittree(StockSpec, RateSpec, TimeSpec, StockOptSpec)` creates the `ittree` structure specifying stock and time information for an implied trinomial tree.

Examples For this example, assume that the interest rate is fixed at 8% annually between the valuation date of the tree (January 1, 2006) until its maturity.

```
Rate = 0.08;
ValuationDate = '01-01-2006';
EndDate = '01-01-2008';

RateSpec = intenvset('StartDates', ValuationDate, 'EndDates', EndDate, ...
    'ValuationDate', ValuationDate, 'Rates', Rate, 'Compounding', -1);
```

To build an ITTTree, create StockSpec, TimeSpec, and StockOptSpec structures.

To create theStockSpec structure:

```

Sigma = 0.20;
AssetPrice = 50;
DividendType = 'cash';
DividendAmounts = [0.50; 0.50; 0.50; 0.50];
ExDividendDates = {'03-Jan-2007'; '01-Apr-2007'; '05-July-2007'; '01-Oct-2007'}

StockSpec = stockspec(Sigma, AssetPrice, DividendType, ...
    DividendAmounts, ExDividendDates)
StockSpec =

    FinObj: 'StockSpec'
    Sigma: 0.2000
    AssetPrice: 50
    DividendType: 'cash'
    DividendAmounts: [4x1 double]
    ExDividendDates: [4x1 double]

```

The syntax for building a TimeSpec structure is TimeSpec = itttimespec(ValuationDate, Maturity, NumPeriods).

Consider building an ITT tree, with a valuation date of January 1, 2006; a maturity date of January 1, 2008; and four time steps.

```

ValuationDate = '01-01-2006';
EndDate = '01-01-2008';
NumPeriods = 4;

TimeSpec = itttimespec(ValuationDate, EndDate, NumPeriods)

TimeSpec =

    FinObj: 'ITTimeSpec'
    ValuationDate: 732678

```

```
        Maturity: 733408
      NumPeriods: 4
        Basis: 0
    EndMonthRule: 1
      tObs: [0 0.5000 1 1.5000 2]
      dObs: [732678 732860 733043 733225 733408]
```

The syntax for building a StockOptSpec structure is [StockOptSpec]
= stockoptspec(OptPrice, Strike, Settle, Maturity, OptSpec).

```
Settle = '01/01/06';

Maturity = ['07/01/06';
           '07/01/06';
           '07/01/06';
           '01/01/07';
           '01/01/07';
           '01/01/07';
           '01/01/07';
           '01/01/07';
           '07/01/07';
           '07/01/07';
           '07/01/07';
           '07/01/07';
           '01/01/08';
           '01/01/08';
           '01/01/08';
           '01/01/08'];

Strike = [113;
         101;
         100;
         88;
         128;
         112;
         100;
         78;
```



```
144;
112;
100;
69;
162;
112;
100;
61];

OptPrice = [           0;
4.807905472659144;
1.306321897011867;
0.048039195057173;
0;
2.310953054191461;
1.421950392866235;
0.020414826276740;
0;
5.091986935627730;
1.346534812295291;
0.005101325584140;
0;
8.047628153217246;
1.219653432150932;
0.001041436654748];

OptSpec = { 'call';
'call';
'put';
'put';
'call';
'call';
'put';
'put';
'call';
'call';
```

```
'put';  
'put';  
'call';  
'call';  
'put';  
'put'];
```

```
StockOptSpec = stockoptspec(OptPrice, Strike, Settle, Maturity, OptSpec)
```

```
StockOptSpec =
```

```
    FinObj: 'StockOptSpec'  
    OptPrice: [16x1 double]  
    Strike: [16x1 double]  
    Settle: 732678  
    Maturity: [16x1 double]  
    OptSpec: {16x1 cell}  
    InterpMethod: 'price'
```

Use the `StockSpec`, `RateSpec`, `TimeSpec`, and `StockOptSpec` structure to create an `ITTree`.

```
ITTree = ittree(StockSpec, RateSpec, TimeSpec, StockOptSpec)
```

```
ITTree =
```

```
    FinObj: 'ITStockTree'  
    StockSpec: [1x1 struct]  
    StockOptSpec: [1x1 struct]  
    TimeSpec: [1x1 struct]  
    RateSpec: [1x1 struct]  
    tObs: [0 0.5000000000000000 1 1.5000000000000000 2]  
    dObs: [732678 732860 733043 733225 733408]  
    STree: {1x5 cell}  
    Probs: {[3x1 double] [3x3 double] [3x5 double] [3x7 double]}
```

See Also

`intenvset`, `itttimespec`, `stockoptspec`, `stockspec`

Purpose Price lookback option from CRR tree

Syntax [Price, PriceTree] = lookbackbycrr(CRRTree, OptSpec, Strike, Settle, ExerciseDates, AmericanOpt)

Arguments

CRRTree	Stock tree structure created by <code>crrtree</code> .
OptSpec	Number of instruments (NINST)-by-1 cell array of strings 'call' or 'put'.
Strike	NINST-by-1 vector of strike price values. Each row is the schedule for one option. To calculate the value of a floating-strike lookback option, specify Strike as NaN.
Settle	NINST-by-1 vector of Settle dates. The settle date for every lookback is set to the valuation date of the stock tree. The lookback argument Settle is ignored.
ExerciseDates	For a European option (AmericanOpt = 0): NINST-by-1 vector 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 (AmericanOpt = 1): NINST-by-2 vector of exercise date boundaries. For each instrument, the option can be exercised on any tree 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

lookbackbycrr

between the valuation date of the stock tree and the single listed exercise date.

AmericanOpt (Optional) If AmericanOpt = 0, NaN, or is unspecified, the option is a European option. If AmericanOpt = 1, the option is an American option.

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

[Price, PriceTree] = lookbackbycrr(CRRTree, OptSpec, Strike, Settle, ExerciseDates, AmericanOpt) calculates the value of fixed- and floating-strike lookback options.

Price is a NINST-by-1 vector of expected option prices at time 0.

PriceTree is a tree structure with a vector of instrument prices at each node.

Examples

Price a lookback option using a CRR binomial tree.

Load the file deriv.mat, which provides CRRTree. The CRRTree structure contains the stock specification and time information needed to price the option.

```
load deriv.mat;
```

Set the required values. Other arguments will use defaults.

```
OptSpec = 'Call';  
Strike = 115;  
Settle = '01-Jan-2003';  
ExerciseDates = '01-Jan-2006';
```

```
Price = lookbackbycrr(CRRTree, OptSpec, Strike, Settle, ...
```

ExerciseDates)

Price =

7.6015

References

Hull, J., and A. White, "Efficient Procedures for Valuing European and American Path-Dependent Options," *Journal of Derivatives*, Fall 1993, pp. 21-31.

See Also

crrtree, instlookback

lookbackbyeqp

Purpose Price lookback option from EQP binomial tree

Syntax [Price, PriceTree] = lookbackbyeqp(EQPtree, OptSpec, Strike, Settle, ExerciseDates, AmericanOpt)

Arguments

EQPtree	Stock tree structure created by eqptree.
OptSpec	Number of instruments (NINST)-by-1 cell array of strings 'call' or 'put'.
Strike	NINST-by-1 vector of strike price values. Each row is the schedule for one option. To calculate the value of a floating-strike lookback option, specify Strike as NaN.
Settle	NINST-by-1 vector of Settle dates. The settle date for every lookback is set to the valuation date of the stock tree. The lookback argument Settle is ignored.
ExerciseDates	For a European option (AmericanOpt = 0): NINST-by-1 vector 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 (AmericanOpt = 1): NINST-by-2 vector of exercise date boundaries. For each instrument, the option can be exercised on any tree 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 valuation date of the stock tree and the single listed exercise date.

AmericanOpt (Optional) If AmericanOpt = 0, NaN, or is unspecified, the option is a European option. If AmericanOpt = 1, the option is an American option.

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

[Price, PriceTree] = lookbackbyeqp(EQPTree, OptSpec, Strike, ExerciseDates, AmericanOpt) calculates the value of fixed- and floating-strike lookback options.

Price is a NINST-by-1 vector of expected option prices at time 0.

PriceTree is a tree structure with a vector of instrument prices at each node.

Examples

Price a lookback option using an EQP equity tree.

Load the file deriv.mat, which provides EQPTree. The EQPTree structure contains the stock specification and time information needed to price the option.

```
load deriv.mat
```

Set the required values. Other arguments will use defaults.

```
OptSpec = 'Call';  
Strike = 115;  
Settle = '01-Jan-2003';  
ExcerciseDates = '01-Jan-2006';
```

```
Price = lookbackbyeqp(EQPTree, OptSpec, Strike, Settle, ...
```

lookbackbyeqp

ExerciseDates)

Price =

8.7941

References

Hull, J., and A. White, "Efficient Procedures for Valuing European and American Path-Dependent Options," *Journal of Derivatives*, Fall 1993, pp. 21-31.

See Also

eqptree, instlookback

Purpose

Price lookback option using an implied trinomial tree (ITT)

Syntax

```
[Price, PriceTree] = lookbackbyitt(ITTTree, OptSpec, Strike,
Settle, ExerciseDates, AmericanOpt)
```

Arguments

ITTTree	Stock tree structure created by itttree.
OptSpec	Number of instruments (NINST)-by-1 cell array of strings 'call' or 'put'.
Strike	NINST-by-1 vector of strike price values. Each row is the schedule for one option. To calculate the value of a floating-strike lookback option, specify Strike as NaN.
Settle	NINST-by-1 vector of Settle dates. The settle date for every lookback option is set to the valuation date of the stock tree. The lookback argument Settle is ignored.
ExerciseDates	<p>For a European option (AmericanOpt = 0):</p> <p>NINST-by-1 vector 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.</p> <p>For an American option (AmericanOpt = 1):</p> <p>NINST-by-2 vector of exercise date boundaries. For each instrument, the option can be exercised on any tree 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</p>

lookbackbyitt

between the valuation date of the stock tree and the single listed exercise date.

AmericanOpt (Optional) If AmericanOpt = 0, NaN, or is unspecified, the option is a European option. If AmericanOpt = 1, the option is an American option.

Note Data arguments for lookbackbyitt 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

[Price, PriceTree] = lookbackbyitt(ITTree, OptSpec, Strike, Settle, ExerciseDates, AmericanOpt) calculates the value of fixed- and floating-strike lookback options.

Price is a NINST-by-1 vector of expected option prices at time 0.

PriceTree is a tree structure with a vector of instrument prices at each node.

Note The Settle date for every lookback option is set to the ValuationDate of the stock tree. The lookback argument, Settle, is ignored.

Examples

Price a lookback option using an ITT equity tree.

Load the file deriv.mat which provides the ITTree. The ITTree structure contains the stock specification and time information needed to price the option.

```
load deriv.mat
```

Set the required values. Other arguments will use defaults.

```
OptSpec = 'Call';  
Strike = 85;  
Settle = '01-Jan-2006';  
ExerciseDates = '01-Jan-2008';  
  
Price = lookbackbyitt(ITTree, OptSpec, Strike, Settle, ExerciseDates)  
  
Price =  
  
0.5426
```

References

Hull, J., and A. White, “Efficient Procedures for Valuing European and American Path-Dependent Options,” *Journal of Derivatives*, Fall 1993, pp. 21-31.

See Also

instlookback, ittree

mkbush

Purpose Create bushy tree

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

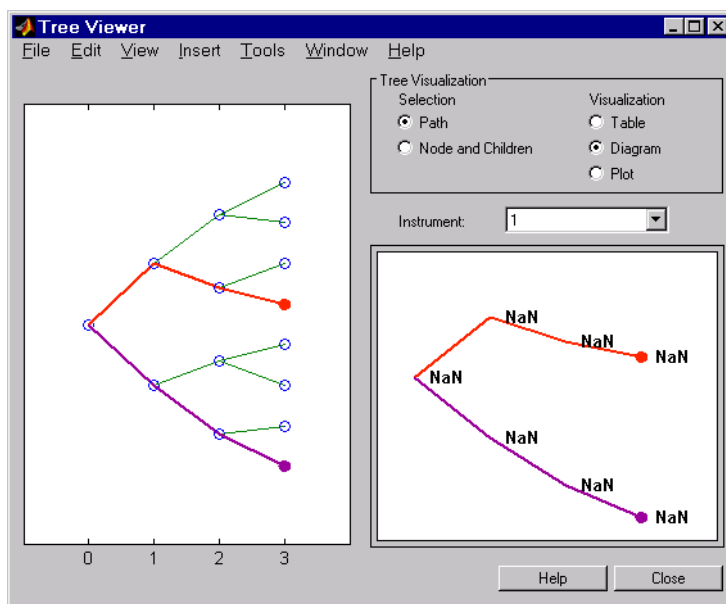
Arguments

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	(Optional) 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	(Optional) 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 binomial tree

Syntax `Tree = mktree(NumLevels, NumPos, NodeVal, IsPriceTree)`

Arguments

<code>NumLevels</code>	Number of time levels of the tree.
<code>NumPos</code>	1-by- <code>NUMLEVELS</code> vector containing the length of the state vectors in each time level.
<code>NodeVal</code>	(Optional) Initial value at each node of the tree. Default = NaN.
<code>IsPriceTree</code>	(Optional) 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 of four time levels with a vector of two elements in each node and each element initialized to NaN.

```
Tree = mktree(4, 2);
```

See Also `treepath`, `treeshape`

Purpose Create recombining trinomial tree

Syntax `TrinTree = mktrintree(NumLevels, NumPos, NumStates, NodeVal)`

Arguments

<code>NumLevels</code>	Number of time levels of the tree.
<code>NumPos</code>	1-by- <code>NUMLEVELS</code> vector containing the length of the state vectors in each time level.
<code>NumStates</code>	1-by- <code>NUMLEVELS</code> vector containing the number of state vectors in each time level.
<code>NodeVal</code>	(Optional) Initial value at each node of the tree. Default = NaN.

Description `TrinTree = mktrintree(NumLevels, NumPos, NumStates, NodeVal)` creates a recombining tree `Tree` with initial values `NodeVal` at each node.

Examples Create a recombining trinomial tree of four time levels with a vector of two elements in each node and each element initialized to NaN.

```
TrinTree = mktrintree(4, [2 2 2 2], [1 3 5 7]);
```

See Also `trintreepath`, `trintreeshape`

mmktbybdt

Purpose Create money-market tree from BDT interest-rate tree

Syntax `MMktTree = mmktbybdt(BDTree)`

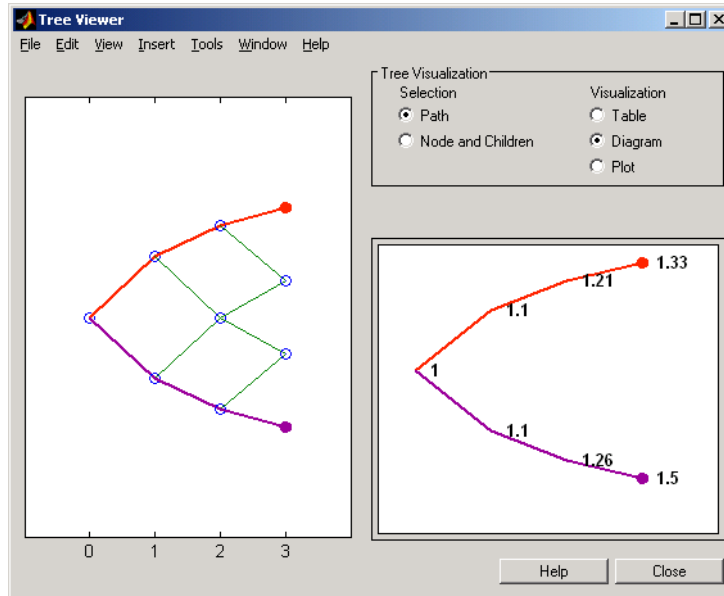
Arguments

`BDTree` Interest-rate tree structure created by `bdttree`.

Description `MMktTree = mmktbybdt(BDTree)` creates a money-market tree from an interest-rate tree structure created by `bdttree`.

Examples

```
load deriv.mat;  
MMktTree = mmktbybdt(BDTree);  
treeviewer(MMktTree)
```



See Also

`bdttree`

mmktbyhjm

Purpose

Create money-market tree from HJM interest-rate tree

Syntax

```
MMktTree = mmktbyhjm(HJMTTree)
```

Arguments

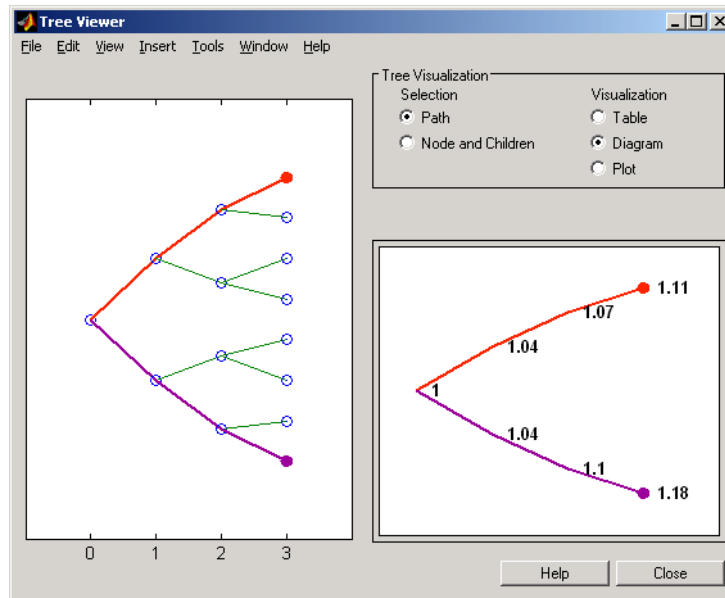
HJMTTree Forward-rate tree structure created by `hjmtree`.

Description

`MMktTree = mmktbyhjm(HJMTTree)` creates a money-market tree from a forward-rate tree structure created by `hjmtree`.

Examples

```
load deriv.mat;  
MMktTree = mmktbyhjm(HJMTTree);  
treeviewer(MMktTree)
```



See Also

hjmtree

optbndbybdt

Purpose Price bond option from 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

BDTree	Forward-rate tree structure created by bdttree.
OptSpec	Number of instruments (NINST)-by-1 cell array of string values 'Call' or 'Put'.
Strike	European option: NINST-by-1 vector of strike price values. 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	<p>NINST-by-1 (European option) or NINST-by-NSTRIKES (Bermuda option) 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.</p> <p>For an American option:</p> <p>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.</p>
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	<p>(Optional) Day-count basis of the instrument. A vector of integers.</p> <ul style="list-style-type: none">• 0 = actual/actual (default)• 1 = 30/360 (SIA)• 2 = actual/360• 3 = actual/365• 4 = 30/360 (PSA)• 5 = 30/360 (ISDA)• 6 = 30/360 (European)• 7 = actual/365 (Japanese)• 8 = actual/actual (ISMA)• 9 = actual/360 (ISMA)• 10 = actual/365 (ISMA)
EndMonthRule	<p>(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.</p>
IssueDate	<p>(Optional) Date when a bond was issued.</p>
FirstCouponDate	<p>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.</p>

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 BDTree. The BDTree structure contains the time and forward-rate information needed to price the bond.

optbndbybdt

```
load deriv.mat;
```

Use `optbndbybdt` to compute the price of the option.

```
Price = optbndbybdt(BDTree, '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(BDTree, '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 from Black-Karasinski interest-rate tree

Syntax

[Price, PriceTree] = optbndbybk(BKTree, OptSpec, Strike, ExerciseDates, AmericanOpt, CouponRate, Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate, StartDate, Face, Options)

Arguments

BkTree	Forward-rate tree structure created by bktree.
OptSpec	Number of instruments (NINST)-by-1 cell array of string values 'Call' or 'Put'.
Strike	<p>European option: NINST-by-1 vector of strike price values.</p> <p>Bermuda option: NINST by number of strikes (NSTRIKES) matrix of strike price values.</p> <p>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.</p> <p>For an American option: NINST-by-1 vector of strike price values for each option.</p>

ExerciseDates	<p>NINST-by-1 (European option) or NINST-by-NSTRIKES (Bermuda option) 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.</p> <p>For an American option:</p> <p>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.</p>
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	<p>(Optional) Day-count basis of the instrument. A vector of integers.</p> <ul style="list-style-type: none">• 0 = actual/actual (default)• 1 = 30/360 (SIA)• 2 = actual/360• 3 = actual/365• 4 = 30/360 (PSA)• 5 = 30/360 (ISDA)• 6 = 30/360 (European)• 7 = actual/365 (Japanese)• 8 = actual/actual (ISMA)• 9 = actual/360 (ISMA)• 10 = actual/365 (ISMA)
EndMonthRule	<p>(Optional) End-of-month rule. A vector. This rule applies only when <code>Maturity</code> 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.</p>
IssueDate	<p>(Optional) Date when a bond was issued.</p>
FirstCouponDate	<p>(Optional) Date when a bond makes its first coupon payment. When <code>FirstCouponDate</code> and <code>LastCouponDate</code> are both specified, <code>FirstCouponDate</code> takes precedence in determining the coupon payment structure.</p>

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 BK tree. The bond argument Settle is ignored.

Description

[Price, PriceTree] = optbndbybk(BKTree, 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 Black-Karasinski 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 BK interest 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, 2006. The settle date for the bond is Jan. 01, 2005, and the maturity date is Jan. 01, 2009.

Load the file deriv.mat, which provides BKTree. The BKTree structure contains the time and forward-rate information needed to price the bond.

```
load deriv.mat;
```

Use `optbndbybk` to compute the price of the option.

```
Price = optbndbybk(BKTree, 'Call', 96, '01-Jan-2006', ...  
'0', '0.04', '01-Jan-2005', '01-Jan-2009')  
Warning: Not all cash flows are aligned with the tree. Result will  
be approximated.
```

```
Price =  
  
0.1512
```

Example 2. Now use `optbndbybk` to compute the price of a put option on the same bond.

```
Price = optbndbybk(BKTree, 'Put', 96, '01-Jan-2006', ...  
'0', '0.04', '01-Jan-2005', '01-Jan-2009')  
  
Warning: Not all cash flows are aligned with the tree. Result will  
be approximated.
```

```
Price =  
  
0.0272
```

See Also `bkprice`, `bktree`, `instoptbnd`

optbndbyhjm

Purpose Price bond option from 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	European option: NINST-by-1 vector of strike price values. 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	<p>NINST-by-1 (European option) or NINST-by-NSTRIKES (Bermuda option) 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.</p> <p>For an American option:</p> <p>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.</p>
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	<p>(Optional) Day-count basis of the instrument. A vector of integers.</p> <ul style="list-style-type: none">• 0 = actual/actual (default)• 1 = 30/360 (SIA)• 2 = actual/360• 3 = actual/365• 4 = 30/360 (PSA)• 5 = 30/360 (ISDA)• 6 = 30/360 (European)• 7 = actual/365 (Japanese)• 8 = actual/actual (ISMA)• 9 = actual/360 (ISMA)• 10 = actual/365 (ISMA)
EndMonthRule	<p>(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.</p>
IssueDate	<p>(Optional) Date when a bond was issued.</p>
FirstCouponDate	<p>(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.</p>

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. The HJMTree structure contains the time and forward-rate information needed to price the bond.

```
load deriv.mat;
```

Use `optbndbyhjm` 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`

Purpose Price bond option from Hull-White interest-rate tree

Syntax [Price, PriceTree] = optbndbyhw(HWTree, OptSpec, Strike, ExerciseDates, AmericanOpt, CouponRate, Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate, StartDate, Face, Options)

Arguments

HWTree	Forward-rate tree structure created by hwtree.
OptSpec	Number of instruments (NINST)-by-1 cell array of string values 'Call' or 'Put'.
Strike	<p>European option: NINST-by-1 vector of strike price values.</p> <p>Bermuda option: NINST by number of strikes (NSTRIKES) matrix of strike price values.</p> <p>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.</p> <p>For an American option:</p> <p>NINST-by-1 vector of strike price values for each option.</p>

ExerciseDates	<p>NINST-by-1 (European option) or NINST-by-NSTRIKES (Bermuda option) 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.</p> <p>For an American option:</p> <p>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.</p>
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	<p>(Optional) Day-count basis of the instrument. A vector of integers.</p> <ul style="list-style-type: none">• 0 = actual/actual (default)• 1 = 30/360 (SIA)• 2 = actual/360• 3 = actual/365• 4 = 30/360 (PSA)• 5 = 30/360 (ISDA)• 6 = 30/360 (European)• 7 = actual/365 (Japanese)• 8 = actual/actual (ISMA)• 9 = actual/360 (ISMA)• 10 = actual/365 (ISMA)
EndMonthRule	<p>(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.</p>
IssueDate	<p>(Optional) Date when a bond was issued.</p>
FirstCouponDate	<p>(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.</p>

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 HW tree. The bond argument Settle is ignored.

Description

[Price, PriceTree] = optbndbyhw(HWTree, 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 Hull-White 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 HW interest 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, 2006. The settle date for the bond is Jan. 01, 2005, and the maturity date is Jan. 01, 2009.

Load the file deriv.mat, which provides HWTree. The HWTree structure contains the time and forward-rate information needed to price the bond.

```
load deriv.mat;
```

Use `optbndbyhw` to compute the price of the option.

```
Price = optbndbyhw(HWTree, 'Call', 96, '01-Jan-2006', ...  
'0', '0.04', '01-Jan-2005', '01-Jan-2009')
```

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

```
Price =
```

```
1.1556
```

Example 2. Now use `optbndbyhw` to compute the price of a put option on the same bond.

```
Price = optbndbyhw(HWTree, 'Put', 96, '01-Jan-2006', ...  
'0', '0.04', '01-Jan-2005', '01-Jan-2009')
```

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

```
Price =
```

```
1.0150
```

See Also

`hwprice`, `hwtree`, `instoptbnd`

optstockbycrr

Purpose Price stock option from CRR tree

Syntax [Price, PriceTree] = optstockbycrr(CRRTree, OptSpec, Strike, Settle, ExerciseDates, AmericanOpt)

Arguments

CRRTree	Stock tree structure created by crrtree.
OptSpec	Number of instruments (NINST)-by-1 cell array of strings 'call' or 'put'.

Note 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	European option: NINST-by-1 vector of strike price values. 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. American option: NINST-by-1 vector of strike price values for each option.
--------	---

Settle	NINST-by-1 vector of settlement or trade dates.
ExerciseDates	NINST-by-1 (European option) or NINST-by-NSTRIKES (Bermuda option) 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

[Price, PriceTree] = optstockbycrr(CRRTree, OptSpec, Strike, Settle, ExerciseDates, AmericanOpt) computes the price of a European, Bermuda, or American stock option.

Price is a NINST-by-1 vector of expected option prices at time 0.

PriceTree is a tree structure with a vector of instrument prices at each node.

Examples

Price a stock option using a CRR binomial tree.

Load the file deriv.mat, which provides CRRTree. The CRRTree structure contains the stock specification and time information needed to price the option.

```
load deriv.mat;
```

Set the required values. Other arguments will use defaults.

```
OptSpec = 'Call';  
Strike = 105;  
Settle = '01-Jan-2003';  
ExerciseDates = '01-Jan-2005';  
  
Price = optstockbycrr(CRRTree, OptSpec, Strike, Settle, ...  
ExerciseDates)  
  
Price =  
  
8.2863
```

See Also

crrtree, instoptstock

Purpose

Price stock option from EQP binomial tree

Syntax

[Price, PriceTree] = optstockbyeqp(EQPtree, OptSpec, Strike, Settle, ExerciseDates, AmericanOpt)

Arguments

EQPtree	Stock tree structure created by eqptree.
OptSpec	Number of instruments (NINST)-by-1 cell array of strings 'call' or 'put'.

Note 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	<p>European option: NINST-by-1 vector of strike price values.</p> <p>Bermuda option: NINST by number of strikes (NSTRIKES) matrix of strike price values.</p> <p>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.</p> <p>American option: NINST-by-1 vector of strike price values for each option.</p>
--------	--

Settle	NINST-by-1 vector of settlement or trade dates.
ExerciseDates	NINST-by-1 (European option) or NINST-by-NSTRIKES (Bermuda option) 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

[Price, PriceTree] = optstockbyeqp(EQPtree, OptSpec, Strike, Settle, ExerciseDates, AmericanOpt) computes the price of a European/Bermuda or American stock option.

Price is a NINST-by-1 vector of expected option prices at time 0.

PriceTree is a tree structure with a vector of instrument prices at each node.

Examples

Price a stock option using an EQP equity tree.

Load the file deriv.mat, which provides EQPtree. The EQPtree structure contains the stock specification and time information needed to price the option.

```
load deriv.mat
```

Set the required values. Other arguments will use defaults.

```
OptSpec = 'Call';  
Strike = 105;  
Settle = '01-Jan-2003';  
ExerciseDates = '01-Jan-2005';  
  
Price = optstockbyeqp(EQPTree, OptSpec, Strike, Settle, ...  
ExerciseDates)  
  
Price =  
  
8.4791
```

See Also

eqptree, instoptstock

optstockbyitt

Purpose Price options on stocks using an implied trinomial tree (ITT)

Syntax [Price, PriceTree] = optstockbyitt(ITTree, OptSpec, Strike, Settle, ExerciseDates, AmericanOpt)

Arguments

ITTree	Stock tree structure created by ittree.
OptSpec	Number of instruments (NINST)-by-1 cell array of strings 'call' or 'put'.

Note The interpretation of the Strike and ExerciseDates arguments depends on 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	European option: NINST-by-1 vector of strike price values. 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. American option: NINST-by-1 vector of strike price values for each option.
Settle	NINST-by-1 vector of settlement or trade dates.

ExerciseDates	<p>For a European or Bermuda option:</p> <p>NINST-by-1 (European option) or NINST-by-NSTRIKES (Bermuda option) 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.</p> <p>For an American option:</p> <p>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.</p>
AmericanOpt	<p>(Optional) If AmericanOpt = 0, NaN, or is unspecified, the option is a European or Bermuda option. If AmericanOpt = 1, the option is an American option.</p>

Note Data arguments for optstockbyitt 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

[Price, PriceTree] = optstockbyitt(ITTree, OptSpec, Strike, Settle, ExerciseDates, AmericanOpt) computes the price of a European/Bermuda or American stock option.

Price is a NINST-by-1 vector of expected option prices at time 0.

PriceTree is a tree structure with a vector of instrument prices at each node.

Note The `Settle` date for every option is set to the `ValuationDate` of the stock tree. The option argument, `Settle`, is ignored.

Examples

Price a stock option using an ITT equity tree.

Load the file `deriv.mat` which provides the `ITTree`. The `ITTree` structure contains the stock specification and time information needed to price the option.

```
load deriv.mat
```

Set the required values. Other arguments will use defaults.

```
OptSpec = 'Put';  
Strike = 80;  
Settle = '01-Jan-2006';  
ExerciseDates = ' 01-Jan-2010 ';
```

```
Price = optstockbyitt(ITTree, OptSpec, Strike, Settle, ExerciseDates)
```

```
Price =
```

```
10.68
```

See Also

`instoptstock`, `itttree`, `stockoptspec`

Purpose

Discount factors from interest rates

Syntax

Usage 1: Interval points are input as times in periodic units.

Disc = rate2disc(Compounding, Rates, EndTimes, StartTimes, Basis, EndMonthRule)

Usage 2: ValuationDate is passed and interval points are input as dates.

[Disc, EndTimes, StartTimes] = rate2disc(Compounding, Rates, EndDates, StartDates, ValuationDate, Basis, EndMonthRule)

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; for example, T = F is one year.

Compounding = 365

Disc = $(1 + Z/F)^{-T}$, where F is the number of days in the basis year and T is a number of days elapsed computed by basis.

Compounding = -1

Disc = $\exp(-T*Z)$, where T is time in years.

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.
Basis	(Optional) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none">• 0 = actual/actual (default)• 1 = 30/360 (SIA)• 2 = actual/360• 3 = actual/365• 4 = 30/360 (PSA)• 5 = 30/360 (ISDA)• 6 = 30/360 (European)• 7 = actual/365 (Japanese)• 8 = actual/actual (ISMA)• 9 = actual/360 (ISMA)

- 10 = actual/365 (ISMA)
- EndMonthRule (Optional) End-of-month rule. A vector. This rule applies only when Maturity is an end-of-month date for a month having 30 or fewer days. 0 = ignore rule, meaning that a bond's coupon payment date is always the same numerical day of the month. 1 = set rule on (default), meaning that a bond's coupon payment date is always the last actual day of the month.

Description

Disc = rate2disc(Compounding, Rates, EndTimes, StartTimes, Basis, EndMonthRule) and [Disc, EndTimes, StartTimes] = rate2disc(Compounding, Rates, EndDates, StartDates, ValuationDate, Basis, EndMonthRule) 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 $T = T_{\text{semi}}/2 * F$, where F is the compounding frequency.

You can specify the investment intervals 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.

Examples

Example 1. Compute discounts from a zero curve at 6 months, 12 months, and 24 months. The times to the cash flows are 1, 2, and 4. You 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 6 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 6 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

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

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; for example, T = F is one year.

Compounding = 365

$Disc = (1 + Z/F)^{-T}$, where F is the number of days in the basis year and T is a number of days elapsed computed by basis.

Compounding = -1

	Disc = $\exp(-T*Z)$, where T is time in years.
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.

ratetimes

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 $T = T_{\text{semi}}/2 * F$, where F is the compounding frequency.

You can specify the investment intervals 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 6, 12, and 24 months. Create a collection of one year forward rates beginning at 0, 6, 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

stockoptspec

Purpose Specify European stock option structure

Syntax `[StockOptSpec] = stockoptspec(OptPrice, Strike, Settle, Maturity, OptSpec, InterpMethod)`

Arguments

<code>OptPrice</code>	NINST-by-1 vector of European option prices.
<code>Strike</code>	NINST-by-1 vector of strike prices.
<code>Settle</code>	Scalar date marking the settlement date.
<code>Maturity</code>	NINST-by-1 vector of maturity dates.
<code>OptSpec</code>	NINST-by-1 cell array of strings 'call' or 'put'.
<code>InterpMethod</code>	(Optional) Method of interpolation to use for option prices. <code>InterpMethod</code> is [<code>{'price'}</code> <code>'vol'</code>]. The default is 'price'. By specifying 'vol', implied volatilities will be used for interpolation purposes. The interpolated values will then be used to calculate the implicit interpolated prices.

Description `[StockOptSpec] = stockoptspec(OptPrice, Strike, Settle, Maturity, OptSpec, InterpMethod)` creates a structure encapsulating the properties of a stock option structure.

Examples Consider the following data quoted from liquid options in the market with varying strikes and maturity. You specify these parameters in MATLAB as follows:

```
Settle = '01/01/06';

Maturity = ['07/01/06';
           '07/01/06';
           '07/01/06'];
```

```
'01/01/07';  
'01/01/07';  
'01/01/07';  
'01/01/07';  
'07/01/07';  
'07/01/07';  
'07/01/07';  
'07/01/07';  
'01/01/08';  
'01/01/08';  
'01/01/08';  
'01/01/08'];
```

```
Strike = [113;
```

```
101;  
100;  
88;  
128;  
112;  
100;  
78;  
144;  
112;  
100;  
69;  
162;  
112;  
100;  
61];
```

```
OptPrice =[                                0;  
4.807905472659144;  
1.306321897011867;  
0.048039195057173;  
0;  
2.310953054191461;  
1.421950392866235;
```

stockoptspec

```
0.020414826276740;  
    0;  
5.091986935627730;  
1.346534812295291;  
0.005101325584140;  
    0;  
8.047628153217246;  
1.219653432150932;  
0.001041436654748];
```

```
OptSpec = { 'call';  
    'call';  
    'put';  
    'put';  
    'call';  
    'call';  
    'put';  
    'put';  
    'call';  
    'call';  
    'put';  
    'put';  
    'call';  
    'call';  
    'put';  
    'put'};
```

```
StockOptSpec = stockoptspec(OptPrice, Strike, Settle, Maturity, OptSpec)
```

```
StockOptSpec =
```

```
    FinObj: 'StockOptSpec'  
    OptPrice: [16x1 double]  
    Strike: [16x1 double]  
    Settle: 732678  
    Maturity: [16x1 double]
```

```
OptSpec: {16x1 cell}  
InterpMethod: 'price'
```

See Also itprice, ittree, stockspect

stockspec

Purpose Create stock structure

Syntax StockSpec = stockspec(Sigma, AssetPrice, DividendType, DividendAmounts, ExDividendDates)

Arguments

Sigma	Decimal annual price volatility of underlying security.
AssetPrice	Scalar representing stock price at time 0.
DividendType	(Optional) Dividend type. Must be cash for actual dollar dividends, constant for constant dividend yield, or continuous for continuous dividend yield.

Note Dividends are assumed to be paid in cash. Noncash dividends (stock) are not allowed.

DividendAmounts	(Optional) Number of dividends (NDIV)-by-1 vector of cash dividends or constant annualized dividend yields, or a scalar representing a continuous annualized dividend yield.
ExDividendDates	(Optional) NDIV-by-1 vector of ex-dividend dates for cash and constant dividend types. Ignored for continuous dividend type.

Description StockSpec = stockspec(Sigma, AssetPrice, DividendType, DividendAmounts, ExDividendDates) creates a MATLAB structure containing the properties of a stock.

Examples

```
Sigma = 0.20;  
AssetPrice = 50;  
DividendType = 'cash';  
DividendAmounts = [0.50; 0.50; 0.50; 0.50];  
ExDividendDates = {'03-Jan-2003'; '01-Apr-2003'; '05-July-2003';  
'01-Oct-2003'}
```

```
StockSpec = stockspec(Sigma, AssetPrice, DividendType, ...  
DividendAmounts, ExDividendDates)  
StockSpec =
```

```
          FinObj: 'StockSpec'  
          Sigma: 0.2000  
    AssetPrice: 50  
    DividendType: 'cash'  
DividendAmounts: [4x1 double]  
ExDividendDates: [4x1 double]
```

See Also

crrprice, crrtree, intenvset

swapbybdt

Purpose Price swap instrument from BDT interest-rate tree

Syntax `[Price, PriceTree, CFTree, SwapRate] = swapbybdt(BDTree,
LegRate, Settle, Maturity, LegReset, Basis, Principal, LegType, Options)`

Arguments

BDTree	Interest-rate tree structure created by <code>bdttree</code> .
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) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none"> • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (PSA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ISMA) • 9 = actual/360 (ISMA) • 10 = actual/365 (ISMA)
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 0. To do this enter CouponRate as NaN.

Description

[Price, PriceTree, CFTree, SwapRate] = swapbybdt(BDTree, 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 a tree structure with a vector of the swap values at each node.

CFTree is a tree structure with a vector of the swap cash flows at each node. This structure contains only NaNs because with binomial recombining trees, cash flows cannot be computed accurately at each node of a tree.

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 arguments 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 arguments 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.mat;
```

Use swapbybdt to compute the price of the swap.

```
Price = swapbybdt(BDTTree, LegRate, Settle, Maturity,...
LegReset, Basis, Principal, LegType)
```

```
Price =
```

```
7.4222
```

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 =
```

```
-1.4211e-014
```

```
PriceTree =
```

```
FinObj: 'BDTPriceTree'
tObs: [0 1 2 3 4]
PTree: {1x5 cell}
```

swapbybdt

```
CFTree =  
  
    FinObj: 'BDTCFTree'  
    tObs: [0 1 2 3 4]  
    CFTree: {[NaN] [NaN NaN] [NaN NaN NaN] [NaN NaN NaN NaN] ...}  
  
SwapRate =  
  
    0.1205
```

See Also [bdttree](#), [capbybdt](#), [cfbybdt](#), [floorbybdt](#)

Purpose Price swap instrument from Black-Karasinski interest-rate tree

Syntax [Price, PriceTree, CFTree, SwapRate] = swapbybk(BKTree, LegRate, Settle, Maturity, LegReset, Basis, Principal, LegType)

Arguments

BKTree	Interest-rate tree structure created by bktree.
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) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none">• 0 = actual/actual (default)• 1 = 30/360 (SIA)• 2 = actual/360• 3 = actual/365• 4 = 30/360 (PSA)• 5 = 30/360 (ISDA)• 6 = 30/360 (European)• 7 = actual/365 (Japanese)• 8 = actual/actual (ISMA)• 9 = actual/360 (ISMA)• 10 = actual/365 (ISMA)
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 BK 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] = swapbybk(BKTree, LegRate, Settle, Maturity, LegReset, Basis, Principal, LegType) computes the price of a swap instrument from a Black-Karasinski 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.

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. The SwapRate output 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 arguments are

- Coupon rate for fixed leg: 0.15 (15%)
- Spread for floating leg: 10 basis points
- Swap settlement date: Jan. 01, 2005
- Swap maturity date: Jan. 01, 2008

Based on the information above, set the required arguments and build the LegRate, LegType, and LegReset matrices.

```
Settle = '01-Jan-2005';
Maturity = '01-Jan-2008';
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 BKTtree included in the MAT-file `deriv.mat`. The BKTtree structure contains the time and forward-rate information needed to price the instrument.

```
load deriv.mat;
```

Use `swapbybk` to compute the price of the swap.

```
Price = swapbybk(BKTtree, LegRate, Settle, Maturity, LegReset,...  
Basis, Principal, LegType)
```

```
Price =
```

```
39.1827
```

Example 2. Using the previous data, calculate the swap rate, which is the coupon rate for the fixed leg such that the swap price at time = 0 is zero.

```
LegRate = [NaN 20];
```

```
[Price, PriceTree, SwapRate] = swapbybk(BKTtree, LegRate, ...  
Settle, Maturity, LegReset, Basis, Principal, LegType)
```

```
Price =
```

```
0
```

```
PriceTree =
```

```
FinObj: 'BKPriceTree'
```

```
tObs: [0 1 2 3 4]
```

```
PTree: {1x5 cell}
```

```
Connect: {[2] [2 3 4] [2 2 3 4 4]}
```

```
Probs: {[3x1 double] [3x3 double] [3x5 double]}
```

```
SwapRate =
```

```
0.0438
```


See Also

bktree, bondbybk, capbybk, fixedbybk, floorbybk

swapbyhjm

Purpose Price swap instrument from HJM interest-rate tree

Syntax [Price, PriceTree, CFTree, SwapRate] = swapbyhjm(HJMTree, LegRate, Settle, Maturity, LegReset, Basis, Principal, LegType)

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) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none"> • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (PSA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ISMA) • 9 = actual/360 (ISMA) • 10 = actual/365 (ISMA)
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 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. The SwapRate output 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 arguments 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 arguments 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. The HJMTree structure contains the time and forward-rate information needed to price the instrument.

```
load deriv.mat;
```

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 =
```

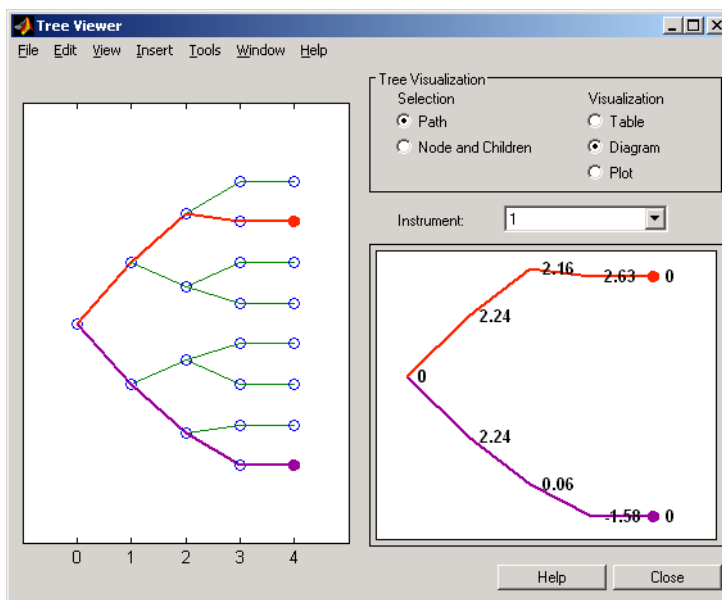
```
FinObj: 'HJMPriceTree'
tObs: [0 1 2 3 4]
PBush: {1x5 cell}
```

```
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 rate and pay floating rate). 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, which is 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

swapbyhw

Purpose Price swap instrument from Hull-White interest-rate tree

Syntax [Price, PriceTree, SwapRate] = swapbyhw(HWTree, LegRate, LegRate, Settle, Maturity, LegReset, Basis, Principal, LegType)

Arguments

HWTree	forward-rate tree structure created by hwtree.
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) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none"> • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (PSA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ISMA) • 9 = actual/360 (ISMA) • 10 = actual/365 (ISMA)
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 HW 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, SwapRate] = swapbyhw(HWTree, LegRate, LegRate, Settle, Maturity, LegReset, Basis, Principal, LegType)` computes the price of a swap instrument from a Hull-White 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.

`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. The `SwapRate` output 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 arguments are

- Coupon rate for fixed leg: 0.06 (6%)
- Spread for floating leg: 20 basis points
- Swap settlement date: Jan. 01, 2005
- Swap maturity date: Jan. 01, 2008

Based on the information above, set the required arguments and build the `LegRate`, `LegType`, and `LegReset` matrices.

```
Settle = '01-Jan-2005';  
Maturity = '01-Jan-2008';  
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 HWTtree included in the MAT-file deriv.mat. The HWTtree structure contains the time and forward-rate information needed to price the instrument.

```
load deriv.mat;
```

Use swapbyhw to compute the price of the swap.

```
[Price, PriceTree, SwapRate] = swapbyhw(HWTtree, LegRate, ...
Settle, Maturity, LegReset, Basis, Principal, LegType)
```

```
Price =
```

```
5.9109
```

```
PriceTree =
```

```
FinObj: 'HWPriceTree'
```

```
tObs: [0 1 2 3 4]
```

```
PTree: {1x5 cell}
```

```
Connect: {[2] [2 3 4] [2 2 3 4 4]}
```

```
Probs: {[3x1 double] [3x3 double] [3x5 double]}
```

```
SwapRate =
```

```
NaN
```

Example 2. Using the previous data, calculate the swap rate, which is the coupon rate for the fixed leg such that the swap price at time = 0 is zero.

```
LegRate = [NaN 20];
```

```
[Price, PriceTree, SwapRate] = swapbyhw(HWTtree, LegRate, ...
Settle, Maturity, LegReset, Basis, Principal, LegType)
```

```
Price =
```

```
1.4211e-014
```

```
PriceTree =  
  
FinObj: 'HWPriceTree'  
  tObs: [0 1 2 3 4]  
  PTree: {1x5 cell}  
  Connect: {[2] [2 3 4] [2 2 3 4 4]}  
  Probs: {[3x1 double] [3x3 double] [3x5 double]}  
  
SwapRate =  
  
0.0438
```

See Also

[bondbyhw](#), [capbyhw](#), [cfbyhw](#), [floorbyhw](#), [fixedbyhw](#), [hwtree](#)

Purpose

Price swap instrument from set of zero curves

Syntax

[Price, SwapRate] = swapbyzero(RateSpec, LegRate, Settle, Maturity, LegReset, Basis, Principal, LegType)

Arguments

RateSpec	A structure containing the properties of an interest-rate structure. See <code>intenvset</code> for information on creating <code>RateSpec</code> .
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) Day-count basis of the instrument. A vector of integers. <ul style="list-style-type: none">• 0 = actual/actual (default)• 1 = 30/360 (SIA)• 2 = actual/360• 3 = actual/365• 4 = 30/360 (PSA)• 5 = 30/360 (ISDA)• 6 = 30/360 (European)• 7 = actual/365 (Japanese)• 8 = actual/actual (ISMA)• 9 = actual/360 (ISMA)• 10 = actual/365 (ISMA)
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 from 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. The SwapRate output 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 arguments 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 arguments 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.mat;
```

Use swapbyzero to compute the price of the swap.

```
Price = swapbyzero(ZeroRateSpec, LegRate, Settle, Maturity,...
    LegReset, Basis, Principal, LegType)
```

swapbyzero

```
Price =  
    3.6923
```

Example 2. Using the previous data, calculate the swap rate, which is 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 Dates from time and frequency

Syntax `Dates = time2date(Settle, Times, Compounding, Basis, EndMonthRule)`

Arguments

Settle	Settlement date. A vector of serial date numbers or date strings.
Times	Vector of times corresponding to the compounding value. Times must be equal to or greater than 0.
Compounding	<p>(Optional) Scalar value representing the rate at which the input zero rates were compounded when annualized. Default = 2. This argument determines the formula for the discount factors:</p> <p>Compounding = 1, 2, 3, 4, 6, 12</p> <p>Disc = $(1 + Z/F)^{-T}$, where F is the compounding frequency, Z is the zero rate, and T is the time in periodic units; for example, $T = F$ is one year.</p> <p>Compounding = 365</p> <p>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.</p> <p>Compounding = -1</p> <p>Disc = $\exp(-T*Z)$, where T is time in years.</p>

time2date

Basis	<p>(Optional) Day-count basis of the instrument. A vector of integers.</p> <ul style="list-style-type: none">• 0 = actual/actual (default)• 1 = 30/360 (SIA)• 2 = actual/360• 3 = actual/365• 4 = 30/360 (PSA)• 5 = 30/360 (ISDA)• 6 = 30/360 (European)• 7 = actual/365 (Japanese)• 8 = actual/actual (ISMA)• 9 = actual/360 (ISMA)• 10 = actual/365 (ISMA)
EndMonthRule	<p>(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.</p>

Description

Dates = time2date(Settle, Times, Compounding, Basis, EndMonthRule) computes dates corresponding to the times occurring beyond the settlement date.

Note To obtain accurate results from this function, the Basis and Dates arguments must be consistent. If the Dates argument contains months that have 31 days, Basis must be one of the values that allow months to contain more than 30 days; for example, Basis = 0, 3, or 7.

The time2date function is the inverse of date2time.

Examples

Show that date2time and time2date are the inverse of each other. First compute the time factors using date2time.

```
Settle = '1-Sep-2002';
Dates = datenum(['31-Aug-2005'; '28-Feb-2006'; '15-Jun-2006';
                '31-Dec-2006']);
Compounding = 2;
Basis = 0;
EndMonthRule = 1;
Times = date2time(Settle, Dates, Compounding, Basis,...
                  EndMonthRule)

Times =

    5.9945
    6.9945
    7.5738
    8.6576
```

Now use the calculated Times in time2date and compare the calculated dates with the original set.

```
Dates_calc = time2date(Settle, Times, Compounding, Basis,...
                       EndMonthRule)

Dates_calc =

    732555
```

time2date

732736

732843

733042

```
datestr(Dates_calc)
```

```
ans =
```

```
31-Aug-2005
```

```
28-Feb-2006
```

```
15-Jun-2006
```

```
31-Dec-2006
```

See Also

`cftimes` in Financial Toolbox documentation

`date2time`, `disc2rate`, `rate2disc`

Purpose Entries from node of recombining binomial tree

Syntax Values = treepath(Tree, BranchList)

Arguments

Tree	Recombining binomial 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 binomial 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(BDTree.FwdTree, [1 2 1])
```

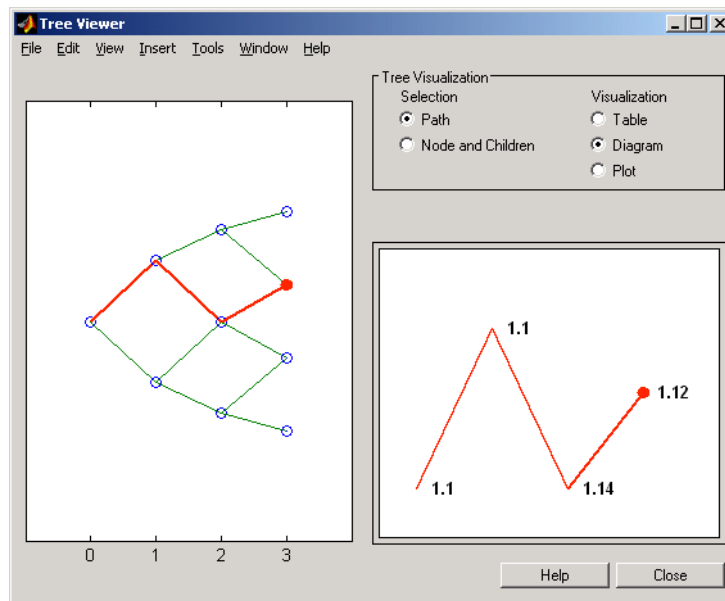
returns the rates at the tree nodes located by taking the up branch, then the down branch, and finally the up branch again.

```
FwdRates =
    1.1000
    1.0979
    1.1377
```

1.1183

You can visualize this with the `treeviewer` function.

```
treeviewer(BDTree)
```



See Also

`mktree`, `treeshape`

Purpose Shape of recombining binomial tree

Syntax [NumLevels, NumPos, IsPriceTree] = treeshape(Tree)

Arguments

Tree Recombining binomial tree.

Description [NumLevels, NumPos, IsPriceTree] = treeshape(Tree) returns information on a recombining binomial 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

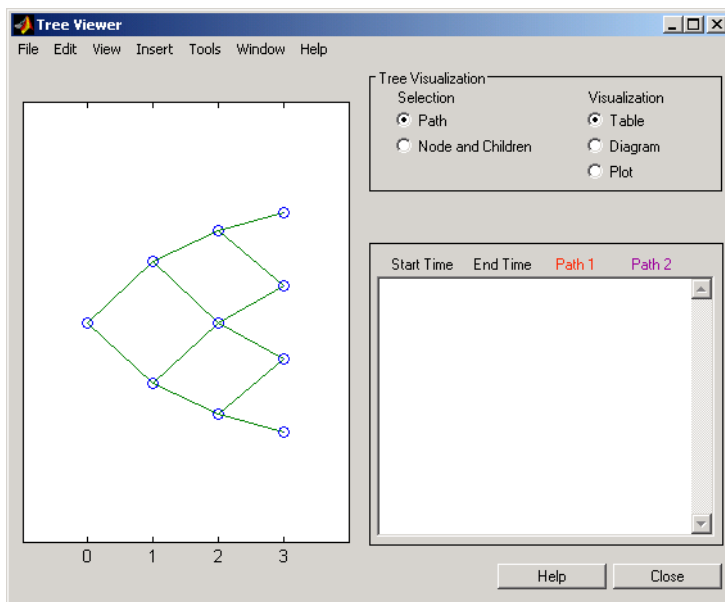
Create a BDT tree by loading the example file.

```
load deriv.mat;
```

With treeviewer you can see the general shape of the BDT interest-rate tree.

```
treeviewer(BDTree)
```

treeshape



With this tree

```
[NumLevels, NumPos, IsPriceTree] = treeshape(BDTree.FwdTree)
```

returns

```
NumLevels =  
    4  
  
NumPos =  
    1    1    1    1  
  
IsPriceTree =  
    0
```

See Also

mktree, treepath

Purpose

Tree information

Syntax

```
treeviewer(Tree)
treeviewer(PriceTree, InstSet)
treeviewer(CFTree, InstSet)
```

Arguments

Tree

Tree can be any of the following types of trees.

Interest-rate trees:

- Black-Derman-Toy (BDTTree)
- Black-Karasinski (BKTree)
- Heath-Jarrow-Morton (HJMTree)
- Hull-White (HWTree)

For information on creating interest-rate trees, see

- `bktree` for information on creating BKTree.
- `bdttree` for information on creating BDTTree.
- `hjmtree` for information on creating HJMTree.
- `hwtree` for information on creating HWTree.

Money market trees:

- Money market tree (MMktTree)

For information on creating money-market trees, see

- `mmktbybdt` for information on creating a money-market tree from a BDT interest-rate tree.
- `mmktbyhjm` for information on creating a money-market tree from an HJM interest-rate tree.

Note Money market trees cannot be created from BK or HW interest-rate trees.

Stock price trees:

- Cox-Ross-Rubinstein (CRRTree)
- Implied Trinomial tree (ITTree)
- Equal probabilities (EQPTree)

For information on creating stock price trees, see

- `crrtree` for information on creating CRRTree.
- `eqptree` for information on creating EQPTree.
- `itttree` for information on creating ITTree.

Cash flow trees:

- Black-Derman-Toy (BDTCFTree)
- Heath-Jarrow-Morton (HJMCFTree)

Cash flow trees are created as outputs from the swap functions `swapbyhjm` and `swapbybdt`.

Note For the function `swapbybdt`, which uses a recombining binomial tree, this structure contains only NaNs because cash flows cannot be accurately calculated at every tree node for floating rate notes.

PriceTree

PriceTree is a Black-Derman-Toy (BDTPriceTree), Black-Karasinski (BKPriceTree), Heath-Jarrow-Morton (HJMPriceTree), Hull-White (HWPriceTree), Cox-Ross-Rubinstein (`crrprice`),

	Equal probabilities (<code>eqpprice</code>), or Implied Trinomial tree (<code>ittprice</code>) tree of instrument prices.
<code>CFTree</code>	<code>CFTree</code> is a tree of swap cash flows. You create cash flow trees when executing the Black-Derman-Toy and Heath-Jarrow-Morton swap functions. (Black-Derman-Toy cash flow trees contain only NaNs.)
<code>InstSet</code>	(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 <code>instadd</code> or as a cell array containing the names of the instruments. To display the names of the instruments, the field <code>Name</code> should exist in <code>InstSet</code> . If <code>InstSet</code> is not passed, <code>treeviewer</code> uses default instruments names (numbers) when displaying prices or cash flows.

Description

`treeviewer(Tree)` displays an interest rate, stock price, 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 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 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 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 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 the nodes along the price or interest rate path shown in the left pane 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 recombining trees, such as BDT, BK and HW, you must click *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 the nodes in the proper order, you are reminded with the message

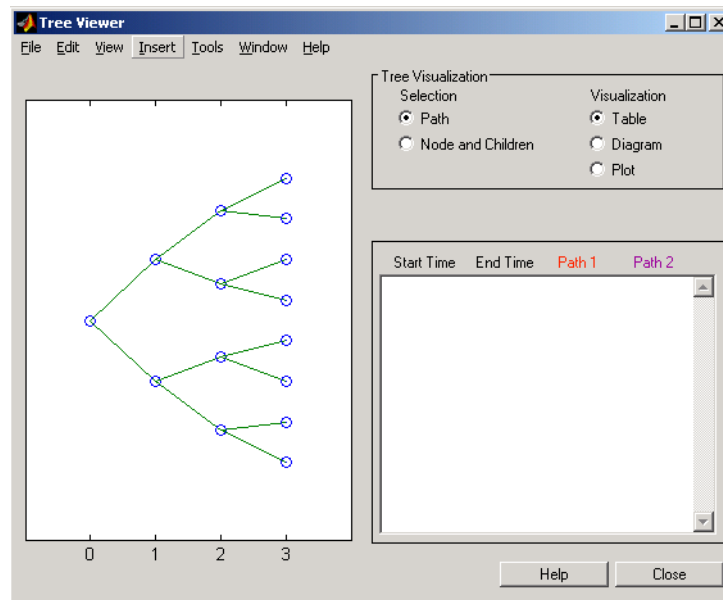
```
Parent of selected node must be selected.
```

Examples

Example 1. Display an HJM Interest-Rate Tree.

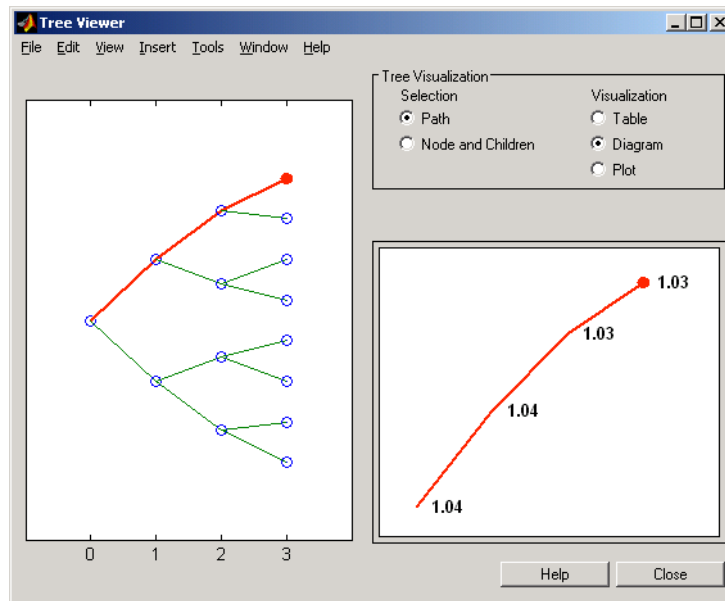
```
load deriv.mat
treeviewer(HJMTree)
```

The treeviewer function displays the structure of an HJM tree in the left pane. The tree visualization in the right pane is blank.



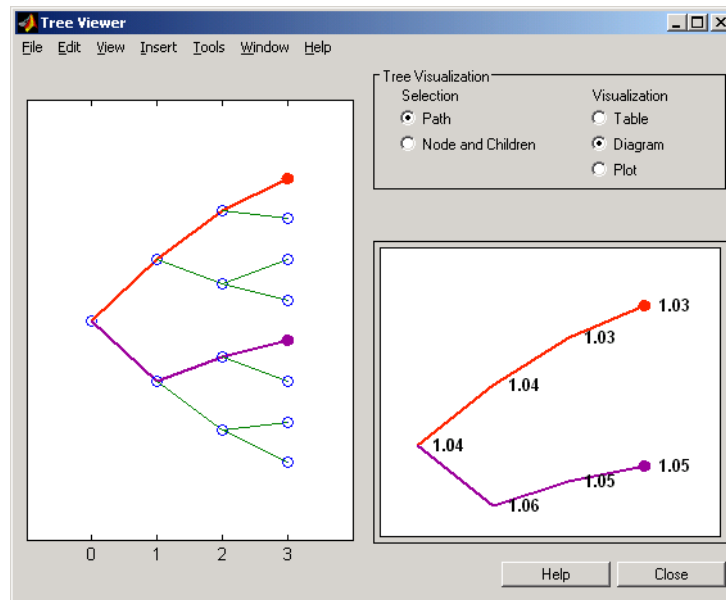
To visualize the actual interest-rate tree, go to the **Tree Visualization** pane and click on **Path** (the default) and **Diagram**. Now, select the first path by clicking on the last node ($t = 3$) of the upper branch.

treeview



Note that the entire upper path is highlighted in red.

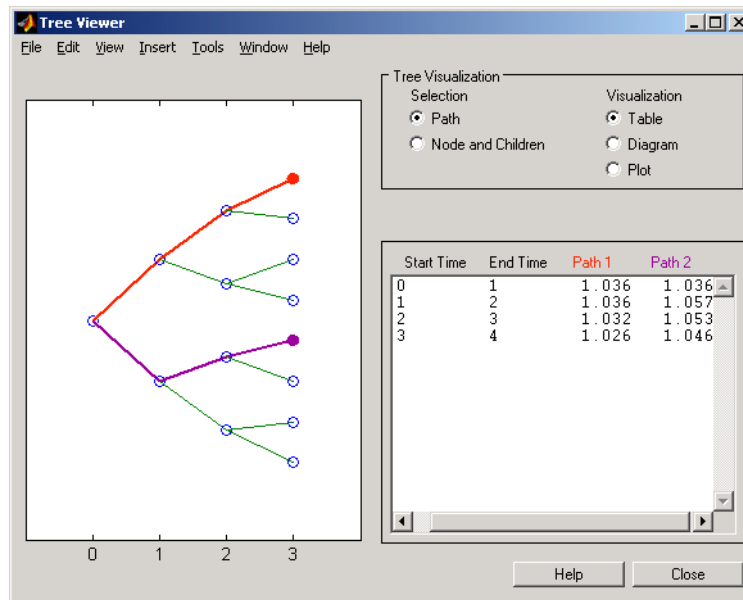
To complete the process, select a second path by clicking on the last node ($t = 3$) of another branch. The second path is highlighted in purple. The final display looks like this.



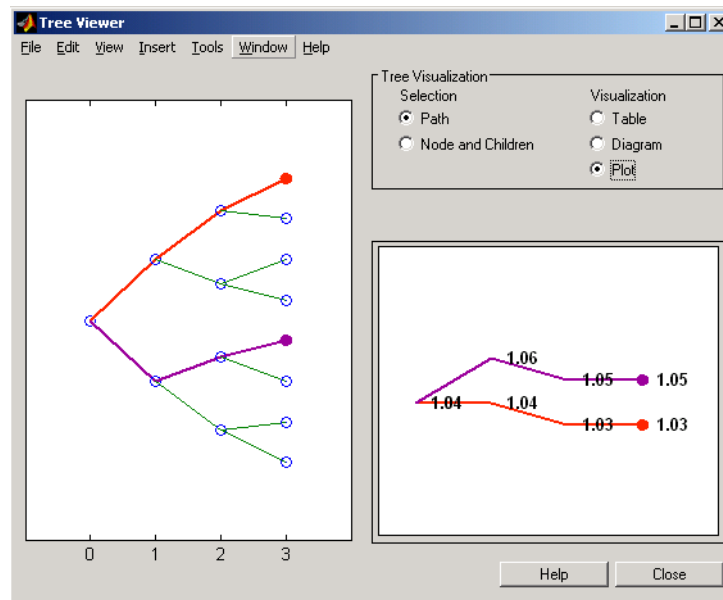
Alternative Forms of Display

The **Tree Visualization** pane 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.

treeview



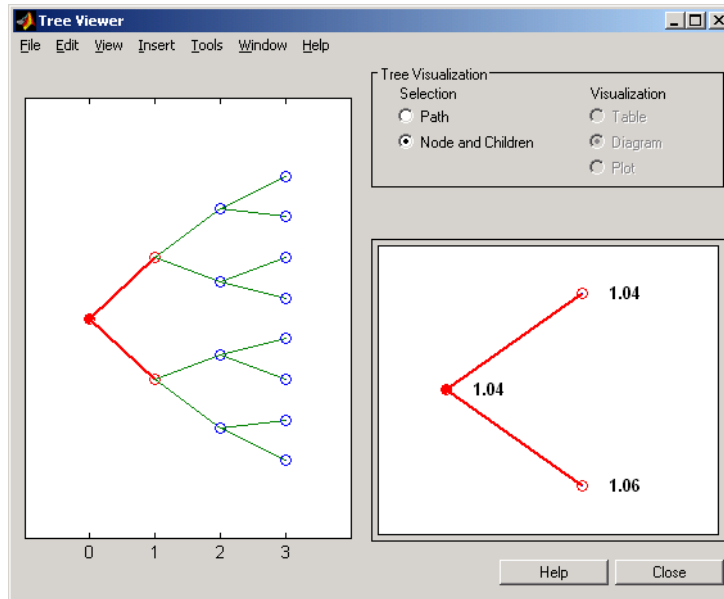
To see a plot of interest rates along the chosen branches, click **Path** and **Plot** in the **Tree Visualization** pane.



Note that with **Plot** selected, rising interest rates are shown on the upper branch and declining interest rates on the lower.

Finally, if you clicked **Node and Children** under **Tree Visualization**, you restrict the data displayed to just the selected parent node and its children.

treeviewer

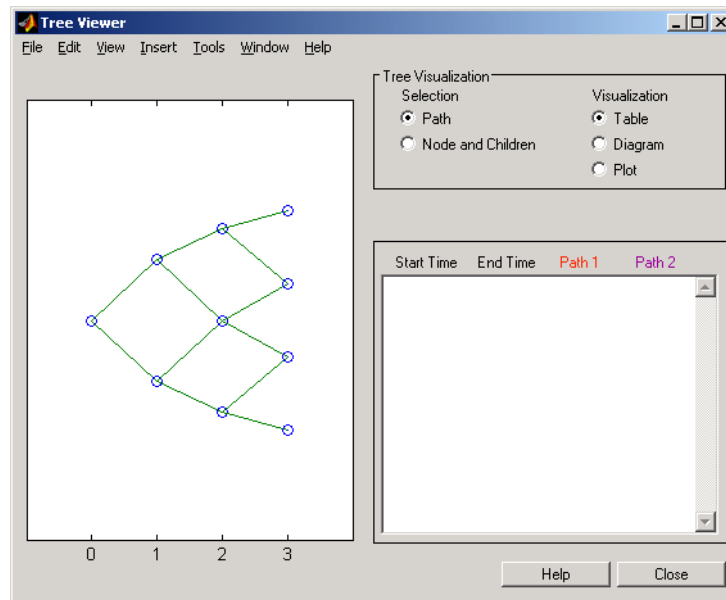


With **Node and Children** selected, the choices under **Visualization** are unavailable.

Example 2. Display a BDT Interest-Rate Tree.

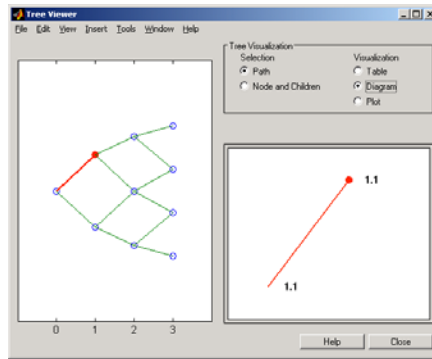
```
load deriv.mat  
treeviewer(BDTTree)
```

The `treeviewer` function displays the structure of a BDT tree in the left pane. The tree visualization in the right pane is blank.

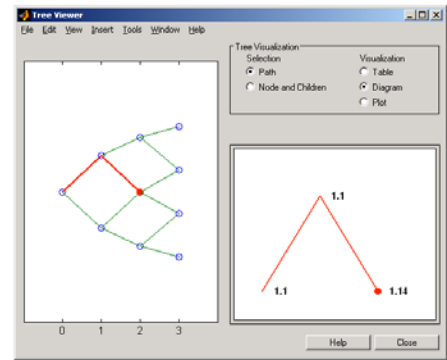


To visualize the actual interest-rate tree, go to the **Tree Visualization** pane and click **Path** (the default) and **Diagram**. Now, select the first path by clicking on the first node of the up branch ($t = 1$). Continue by clicking the down branch at the next node ($t = 2$). The two figures below show the treeviewer path diagrams for these selections.

treeview



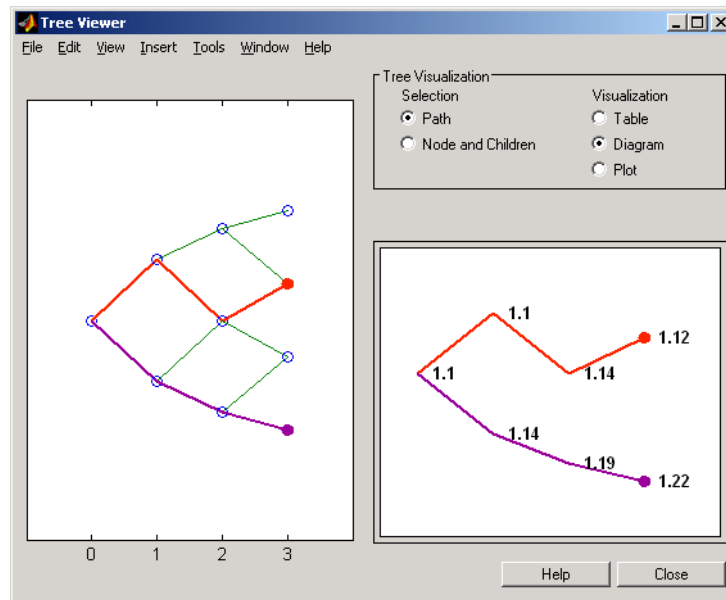
$t = 1$



$t = 2$

Continue clicking all nodes in succession until you reach the end of the branch. Note that the entire path you have selected is highlighted in red.

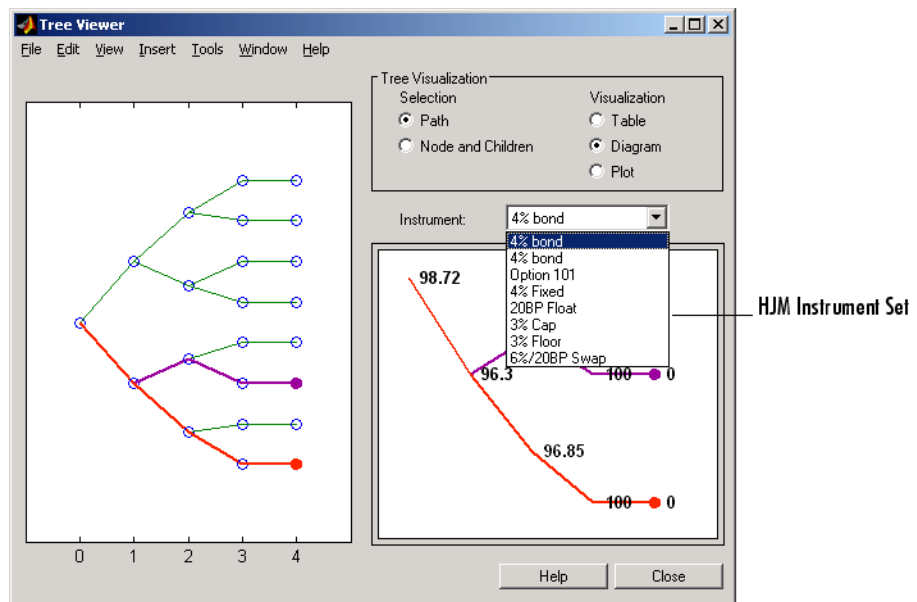
Select a second path by clicking the first node of the lower branch ($t = 1$). Continue clicking lower nodes as you did on the first branch. Note that the second branch is highlighted in purple. The final display looks like this.



Example 3. Display an HJM Price Tree for Named Instruments.

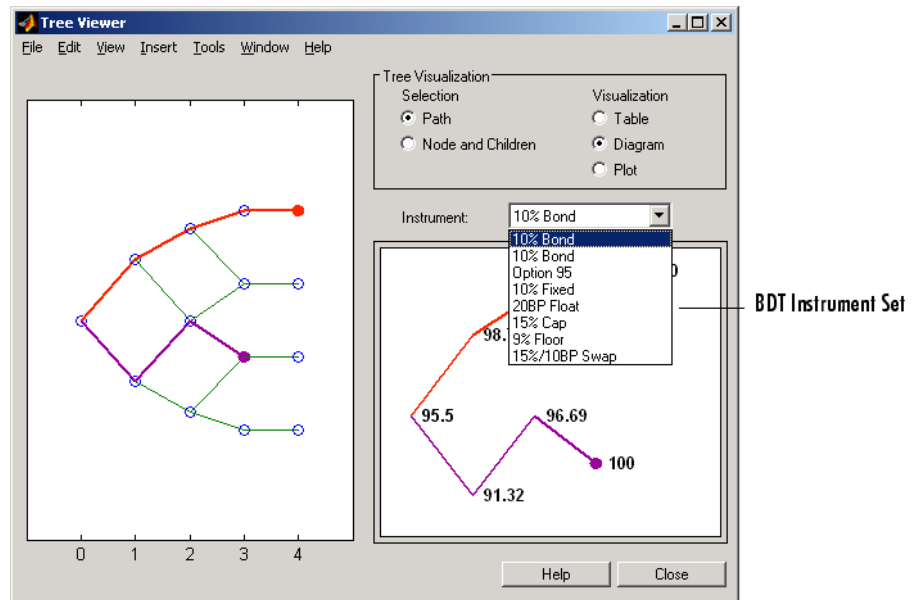
```
load deriv.mat
[Price, PriceTree] = hjmprice(HJMTree, HJMInstSet);
treeviewer(PriceTree, HJMInstSet)
```

treeview



Example 4. Display a BDT Price Tree for Named Instruments.

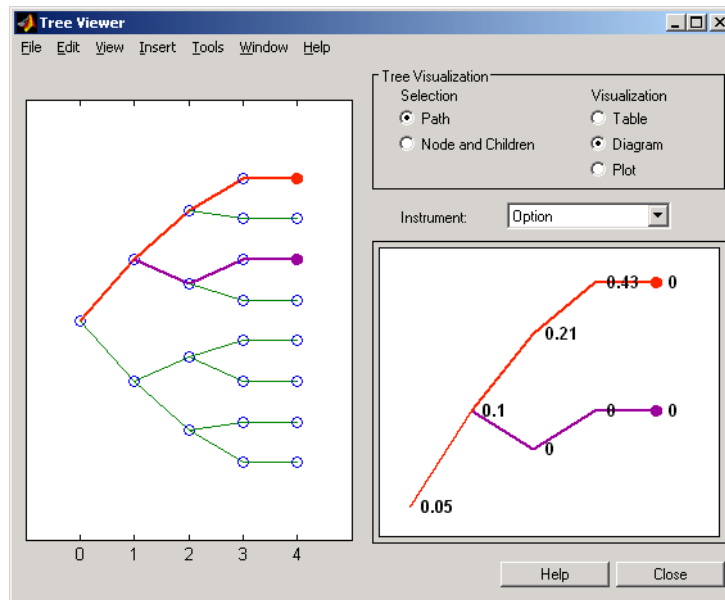
```
load deriv.mat  
[Price, PriceTree] = bdtprice(BDTree, BDTInstSet);  
treeview(PriceTree, BDTInstSet)
```



Example 5. Display an HJM Price Tree with Renamed Instruments.

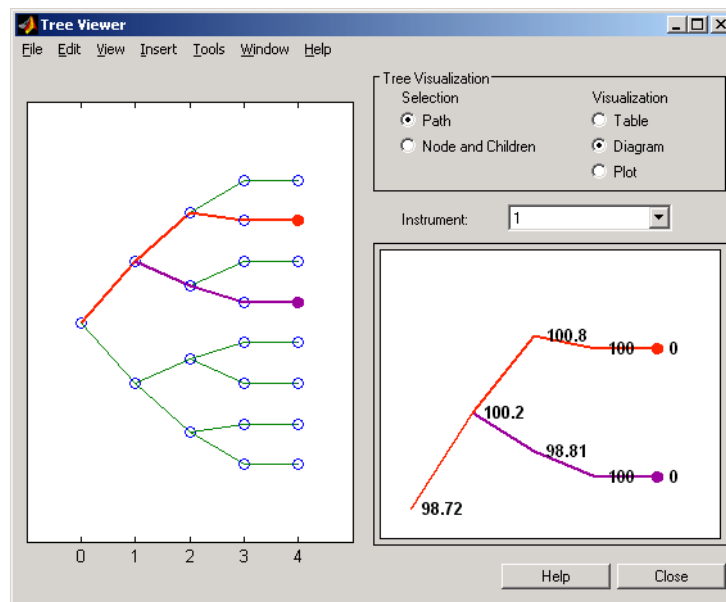
```
load deriv.mat
[Price, PriceTree] = hjmprice(HJMTree, HJMInstSet);
Names = {'Bond1', 'Bond2', 'Option', 'Fixed', 'Float', 'Cap', ...
        'Floor', 'Swap'};
treeviewer(PriceTree, Names)
```

treeview



Example 6. Display an HJM Price Tree Using Default Instrument Names (Numbers).

```
load deriv.mat
[Price, PriceTree] = hjmprice(HJMTree, HJMInstSet);
treeview(PriceTree)
```


**See Also**

bdttree, bktree, crrtree, eqptreehjmtree, hwtree, instadd, itttreemktbybdt, mmktbyhjm, swapbybdt, swapbyhjm

trintreepath

Purpose Entries from node of recombining trinomial tree

Syntax `Values = trintreepath(TrinTree, BranchList)`

Arguments

`TrinTree` Recombining price or interest-rate trinomial tree.

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

Description

`Values = trintreepath(TrinTree, BranchList)` extracts entries of a node of a recombining trinomial tree. The node path is described by the sequence of branchings taken, starting at the root. The top branch is number 1, the middle branch is 2, and the bottom branch is 3. Set the branch sequence to 0 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 Hull-White tree by loading the example file.

```
load deriv.mat;
```

Then, for example

```
FwdRates = trintreepath(HWTree, [1 2 3])
```

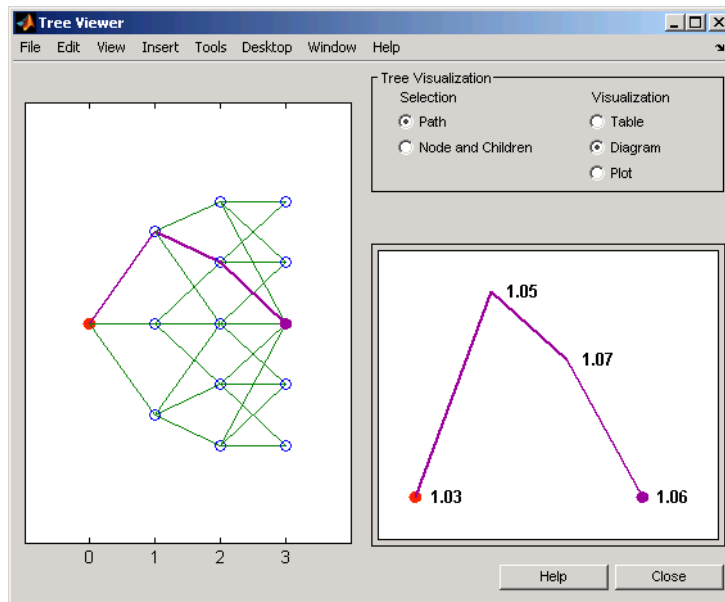
returns the rates at the tree nodes located by starting at 0, taking the up branch at the first node, the middle branch at the second node, and finally the bottom branch at the third node.

```
FwdRates =
```

```
1.0279
1.0528
1.0652
1.0591
```

You can visualize this with the `treeviewer` function.

```
treeviewer(HWTree)
```



See Also `mktrintree`, `trintreeshape`

trintreeshape

Purpose Shape of recombining trinomial tree

Syntax `[NumLevels, NumPos, NumStates] = trintreeshape(TrinTree)`

Arguments

`TrinTree` Recombining price or interest-rate trinomial tree.

Description `[NumLevels, NumPos, NumStates] = trintreeshape(TrinTree)` returns information on a recombining trinomial 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.

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

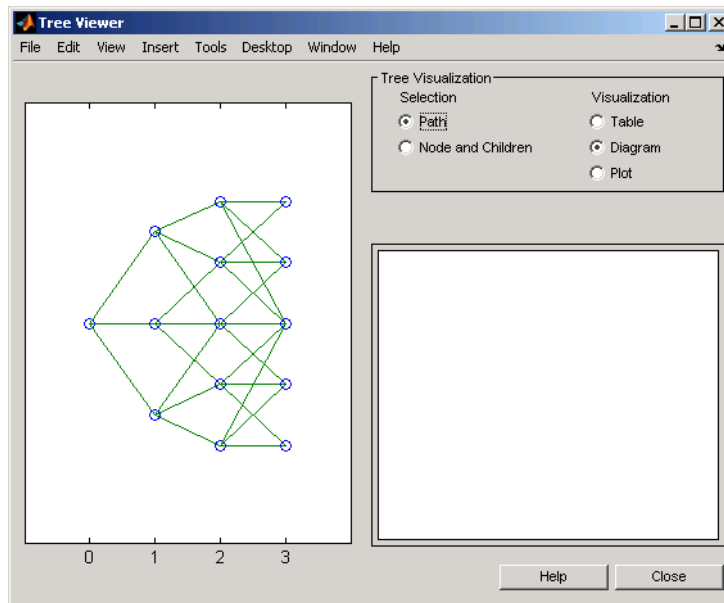
Examples

Create a Hull-White tree by loading the example file.

```
load deriv.mat;
```

With `treeviewer` you can see the general shape of the HW interest-rate tree.

```
treeviewer(HWTTree)
```



With this tree

```
[NumLevels, NumPos, NumStates] = trintreeshape(HWTTree)
```

returns

```
NumLevels =
    4
```

```
NumPos =
    1    1    1    1
```

```
NumStates =
    1    3    5    5
```

See Also

mktrintree, trintreepath

Derivatives Pricing Options

Pricing Options Structure (p. A-2)

Using the pricing Options structure

Customizing the Structure (p. A-5)

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

Pricing Options Structure

The MATLAB Options structure provides additional input to most pricing functions. 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.
- Tells pricing functions which method to use in pricing barrier options.

The pricing options structure is primarily used in the pricing of interest-rate-based financial derivatives. However, the `BarrierMethod` field in the structure allows you to use it in pricing equity barrier options as well.

You provide pricing options in an optional `Options` argument passed to a pricing function. (See, for example, `bondbyhjm`, `bdtprice`, `barrierbycrr`, `barrierbyeqp`, or `barrierbyitt`.)

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'
    BarrierMethod: 'unenhanced'
```

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

Diagnostics Field

Diagnostics indicates whether additional information is displayed if the 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 tree generated for pricing purposes.

Warnings Field

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 Field

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 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 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 a tree grows exponentially with the number of tree dates. Consequently, setting ConstRate 'off' dramatically increases the memory and processor demands on the computer.

BarrierMethod Field

When using binomial trees to price barrier options, you may require a large number of tree steps to achieve an accurate result when tree nodes do not align with the barrier level. With the BarrierMethod field, the toolbox

provides an enhancement method that improves the accuracy of the results without having to use large trees.

The `BarrierMethod` field can be set to 'unenhanced' (default) or 'interp'. If you specify 'unenhanced', no correction calculation is used. Otherwise, if you specify 'interp', the toolbox provides an enhanced valuation by interpolating between nodes on barrier boundaries.

You specify the barrier method in the last input argument, `Options`, of the functions `barrierbycrr`, `barrierbyeqp`, `crrprice`, or `eqpprice`. `Options` is a structure that you create with the function `derivset`. Using `derivset`, you specify whether to use the enhanced or the unenhanced method.

For more information about this algorithm, see Derman, E., I. Kani, D. Ergener and I. Bardhan, "Enhanced Numerical Methods for Options with Barriers," *Financial Analysts Journal*, (Nov. - Dec. 1995), pp. 65-74.

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 `Diagnostics` 'on'.

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

Options =

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

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 January 1, 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

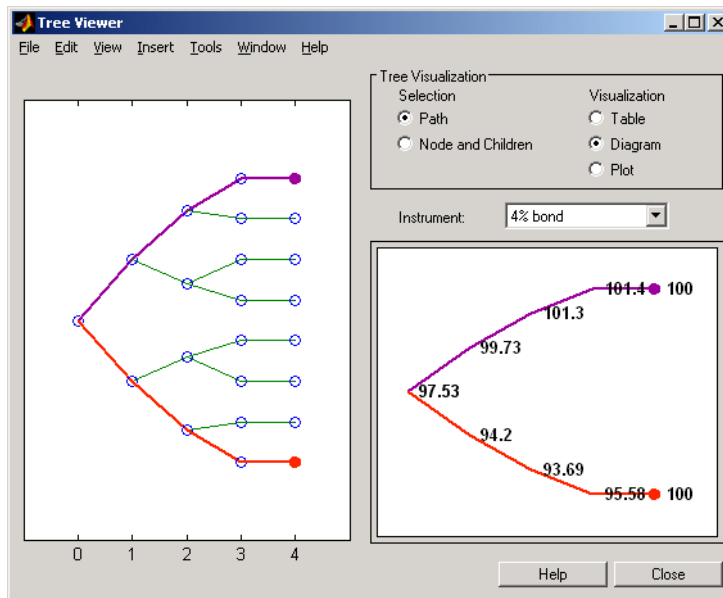
size(BondPriceTreeNoCR.PBush{end})

ans =

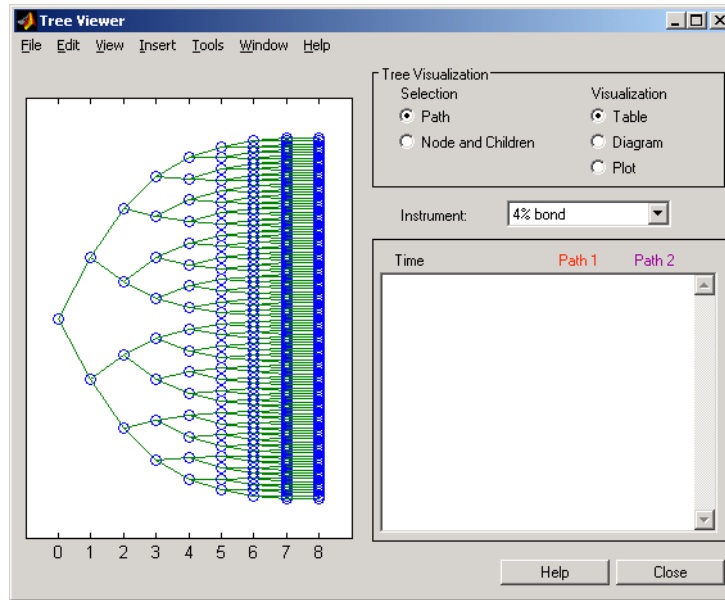
1 128

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

treeviewer(BondPriceTree, BondPort)



treeviewer(BondPriceTreeNoCR, BondPort)



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

Bibliography

Black-Derman-Toy (BDT) Modeling
(p. B-2)

References for the
Black-Derman-Toy interest-rate
model

Heath-Jarrow-Morton (HJM)
Modeling (p. B-3)

References for the
Heath-Jarrow-Morton modeling,
used extensively in Financial
Derivatives Toolbox

Hull-White (HW) and
Black-Karasinski (BK) Modeling
(p. B-4)

References for the Hull-White
model and its Black-Karasinski
modification

Cox-Ross-Rubinstein (CRR)
Modeling (p. B-5)

References for the
Cox-Ross-Rubinstein model

Implied Trinomial Tree (ITT)
Modeling (p. B-6)

References for the Implied Trinomial
Tree model

Equal Probabilities Tree (EQP)
Modeling (p. B-7)

References for the Equal
Probabilities model

Financial Derivatives (p. B-8)

General references for financial
derivatives

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 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.

Hull-White (HW) and Black-Karasinski (BK) Modeling

A description of the Hull-White model and its Black-Karasinski modification can be found in:

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

You can find additional information about the Hull-White single-factor model used in this toolbox in these papers:

Hull, J., and A. White, "Numerical Procedures for Implementing Term Structure Models I: Single-Factor Models," *Journal of Derivatives*, 1994.

Hull, J., and A. White, "Using Hull-White Interest Rate Trees," *Journal of Derivatives*, 1996.

Cox-Ross-Rubinstein (CRR) Modeling

To learn about the Cox-Ross-Rubinstein model, see:

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

Implied Trinomial Tree (ITT) Modeling

To learn about the Implied Trinomial Tree model, see:

Chriss, Neil A., E. Derman, and I. Kani, "Implied trinomial trees of the volatility smile," *Journal of Derivatives*, 1996.

Equal Probabilities Tree (EQP) Modeling

To learn about the Equal Probabilities model, see:

Chriss, Neil A., *Black Scholes and Beyond: Option Pricing Models*,
McGraw-Hill, 1996, ISBN 0-7863-1025-1.

Financial Derivatives

You can find information on the creation of financial derivatives and their role in the marketplace in numerous sources. Among those consulted in the development of 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.

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

Examples

Use this list to find examples in the documentation.

Instrument Portfolio Examples

“Creating New Instruments or Properties” on page 1-9

“instfind Examples” on page 1-12

“instselect Examples” on page 1-15

Interest Rate Environment Examples

“Calculating Discount Factors from Rates” on page 2-9

“Calculating Rates from Discounts” on page 2-13

“Spot Curve to Forward Curve Conversion” on page 2-14

“Example: Pricing a Portfolio of Instruments” on page 2-24

“Example: Sensitivities and Prices” on page 2-26

HJM Examples

“Specifying the Volatility Model (VolSpec)” on page 2-31

“Creating an HJM Tree” on page 2-37

“HJM Pricing Example” on page 2-51

Volatility Modeling

“HJM Volatility Specification Example” on page 2-31

BDT Examples

“BDT Volatility Specification Example” on page 2-33

“Creating a BDT Tree” on page 2-37

“BDT Tree Structure” on page 2-43

“BDT Pricing Example” on page 2-53

Rate Specification Creation

“Rate Specification Creation Example” on page 2-34

Time Specification

“HJM Time Specification Example” on page 2-36

“Creating a BDT Time Specification” on page 2-36

Sensitivity

“HJM Sensitivities Example” on page 2-59

“BDT Sensitivities Example” on page 2-60

“CRR Sensitivities Example” on page 3-39

“ITT Sensitivities Example” on page 3-40

Treeviewer Examples

“Valuation Date Prices” on page 2-67

“Additional Observation Times” on page 2-69

Creating Equity Derivatives

“Stock Structure Example Using a Binary Tree” on page 3-4

“TimeSpec Example Using a Binary Tree” on page 3-6

“Examples of Binary Tree Creation” on page 3-7

“Stock Structure Example Using an Implied Trinomial Tree” on page 3-10

“TimeSpec Example Using an Implied Trinomial Tree” on page 3-11

“Option Stock Structure Example Using an Implied Trinomial Tree” on page 3-12

Pricing Equity Derivatives

“Computing Prices Using CRR” on page 3-28

“Computing Prices Using EQP” on page 3-30

“Computing Prices Using ITT” on page 3-32

Hedging Examples

“Maintaining Existing Allocations” on page 4-5

“Partially Hedged Portfolio” on page 4-6

“Fully Hedged Portfolio” on page 4-8

“Minimizing Portfolio Sensitivities” on page 4-9

“Self-Financing Hedges with hedgeslf” on page 4-11

“Specifying Constraints with ConSet” on page 4-15

Hedging with Constrained Portfolios

“Example: Fully Hedged Portfolio” on page 4-20

“Example: Minimize Portfolio Sensitivities” on page 4-23

“Example: Under-Determined System” on page 4-24

“Example: Portfolio Constraints with hedgeslf” on page 4-25

American option

An option that can be exercised any time until its expiration date. Contrast with **European option** on page Glossary-3.

arbitrary cash flow instrument

A set of generic cash flow amounts for which a price needs to be established.

Asian option

An option whose payoff depends upon the average price of the underlying asset over a certain period of time.

barrier option

An option that is activated or deactivated only if the price of the underlying asset crosses a barrier. See also **knock-in** on page Glossary-5 and **knock-out** on page Glossary-5. If the option fails to execute, the seller may pay to the purchaser a predetermined **rebate** on page Glossary-7.

Bermuda option

An option that can be exercised on predetermined dates only, usually every month. See also **American option** on page Glossary-1 and **European option** on page Glossary-3.

beta

The price volatility of a financial instrument relative to the price volatility of a market or index as a whole. Beta is most commonly used with respect to equities. A high-beta instrument is riskier than a low-beta instrument.

binomial model

A method in which the probability over time of each possible price or rate follows a binomial distribution. The basic assumption is that prices or rates can move to only two values (one higher and one lower) over any short time period. See also **trinomial model** on page Glossary-8.

Black-Derman-Toy (BDT) model

A model for pricing interest rate derivatives where all security prices and rates depend upon the short rate (annualized one-period interest rate).

bond

A long-term debt security with fixed interest payments and fixed maturity date.

bond option

The right to sell a bond back to the issuer (put) or to redeem a bond from its current owner (call) at a specific price and on a specific date.

bushy tree

A tree of prices or interest rates in which the number of branches increases exponentially relative to observation times; branches never recombine. Opposite of a **recombining tree** on page Glossary-7.

call

1. An option to buy a certain quantity of a stock or commodity for a specified price within a specified time. See also **put** on page Glossary-6.
2. A demand to submit bonds to the issuer for redemption before the maturity date.

callable bond

A bond that allows the issuer to buy back the bond at a predetermined price at specified future dates. The bond contains an embedded call option; that is, the holder has sold a call option to the issuer. See also **puttable bond** on page Glossary-7.

cap

Interest-rate option that guarantees that the rate on a floating-rate loan will not exceed a certain level.

compound option

An option on an option, such as a call on a call, a put on a put, a call on a put, or a put on a call.

delta

The rate of change of the price of a derivative security relative to the price of the underlying asset; that is, the first derivative of the curve that relates the price of the derivative to the price of the underlying security.

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.

dollar sensitivity

Sensitivity reported as a dollar price change instead of a percentage price change.

down-and-in

A type of **barrier option** on page Glossary-1 that becomes active if the barrier is reached from above. See also **knock-in** on page Glossary-5.

down-and-out

A type of **barrier option** on page Glossary-1 that becomes deactivated if the barrier is reached from above. See also **knock-out** on page Glossary-5.

European option

An option that can be exercised only on its expiration date. Contrast with **American option** on page Glossary-1.

ex-dividend date

Date when a declared dividend belongs to the seller rather than the buyer.

exercise price

The price set for buying an asset (call) or selling an asset (put). The strike price.

exotic option

Any nonstandard option. Opposite of **vanilla option** on page Glossary-9.

fixed lookback option

Strike price is fixed at purchase. The underlying is priced at its highest or lowest level, depending whether it is a call or put, during the life of the option rather than expiring at market.

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 lookback option

Strike price is fixed at maturity. For a call, the price is fixed at the lowest price during the life of the option; for a put it is fixed at the highest price.

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; that is, 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** on page Glossary-2.

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.

knock-in

A **barrier option** on page Glossary-1 that is activated when the price of the underlying asset achieves a designated target. There are two types: **up-and-in** on page Glossary-8 and **down-and-in** on page Glossary-3.

knock-out

A **barrier option** on page Glossary-1 that is deactivated when the price of the underlying asset achieves a designated target. There are two types: **up-and-out** on page Glossary-8 and **down-and-out** on page Glossary-3.

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.

lookback option

An option that reduces uncertainties associated with the timing of market entry. Lookback options can be either **fixed lookback option** on page Glossary-4 and **floating lookback option** on page Glossary-4.

mean reversion

The tendency of a variable to return to its mean value after reaching a point of excessive positive or negative valuation relative to the mean.

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** on page Glossary-7 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 also **call** on page Glossary-2.

puttable bond

A bond that allows the holder to redeem the bond at a predetermined price at specified future dates. The bond contains an embedded put option; that is, the holder has bought a put option. See also **callable bond** on page Glossary-2.

rate specification

A MATLAB structure that holds all information needed to identify completely the evolution of interest rates.

rebate

A predetermined amount of money paid to the purchaser of a **barrier option** on page Glossary-1 if the option fails to execute.

recombining tree

A tree of prices or interest rates whose branches recombine over time. Opposite of a **bushy tree** on page Glossary-2.

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. See also **dollar sensitivity** on page Glossary-3.

short rate

The annualized one-period interest rate.

spot curve, spot yield curve

See **zero curve, zero-coupon yield curve** on page Glossary-9.

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** on page Glossary-4.

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.

trinomial model

A method in which the basic assumption is that prices or rates can move to one of three possible values over any short time period. At any time step the price or rate direction can be upward, neutral, or downward. See also **binomial model** on page Glossary-1.

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.

up-and-in

A type of **barrier option** on page Glossary-1 that becomes active if the barrier is reached from below. See also **knock-in** on page Glossary-5.

up-and-out

A type of **barrier option** on page Glossary-1 that becomes deactivated if the barrier is reached from below. See also **knock-out** on page Glossary-5.

vanilla option

A common option, such as a put or call. Opposite of **exotic option** on page Glossary-4.

vanilla swap

A **swap** on page Glossary-8 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.

A

- arbitrary cash flow instruments 2-3
- Asian option
 - defined 3-22
- Asian options
 - fixed and floating strike, by CRR 6-3
 - fixed and floating strike, by EQP 6-6
 - fixed and floating strike, by ITT 6-9
- asianbycrr 6-2
- asianbyeqp 6-5
- asianbyitt 6-8
- average price options
 - by CRR 6-3
 - by EQP 6-6
 - by ITT 6-9
- average strike options
 - by CRR 6-3
 - by EQP 6-6
 - by ITT 6-9

B

- bank format 4-4
- barrier option
 - defined 3-23
 - types of 3-23
- barrierbycrr 6-11
- barrierbyeqp 6-14
- barrierbyitt 6-17
- BDT model 2-4
- bdtprice 6-20
- bdtzens 6-23
- bdttimespec 6-26
- bdttree 6-28
 - input arguments 2-29
- bdtvolspec 6-30
 - forms of volatility 2-30
- Bermuda option
 - bond 2-2
 - stock 3-25

- binomial trees 2-5
- BK model 2-4
- bkprice 6-32
- bksens 6-35
- bktimespec 6-38
- bktree 6-40
- bkvolspec 6-43
- Black-Derman-Toy (BDT) model 2-28
- Black-Derman-Toy tree 2-50
- Black-Karasinski (BK) model 2-28
- bond
 - defined 2-2
- bondbybdt 6-45
- bondbybk 6-49
- bondbyhjm 6-53
- bondbyhw 6-57
- bondbyzero 6-61
- bushpath 6-64
 - example 2-42
- bushshape 6-66
- bushy trees 2-5

C

- cap, defined 2-3
- capbybdt 6-69
- capbybk 6-73
- capbyhjm 6-76
- capbyhw 6-79
- cfbybdt 6-82
- cfbybk 6-86
- cfbyhjm 6-90
- cfbyhw 6-94
- cfbyzero 6-98
- classfin 6-100
- compound option
 - defined 3-24
- compoundbyeqp 6-105
- compoundbyitt 6-108
- constraints 4-22

- dependent 4-22
- inconsistent 4-25
- constructor 1-8
- convbyzero 6-127
- couponbycrr 6-102
- CRR and EQP
 - differences 3-20
- CRR model description 3-2
- crrprice 6-111
- crrsens 6-114
- crrtimespec 6-117
- crrtree 6-119
- cvtree 6-123

D

- date2time 6-127
- datedisp 6-130
- delta 2-26
 - defined 4-3
- dependent constraints 4-22
- deriv.mat 2-6
- derivget 6-132
- derivset 6-134
- deterministic model 2-23
- differences between CRR and EQP 3-20
- disc2rate 6-137
 - purpose 2-9
 - syntax 2-13
- discount factors 2-9
- discrete time models 3-2
- dollar sensitivities
 - from interest-rate models 2-58
 - from interest-rate term structure 2-26
 - from stock trees 3-38

E

- EQP model description 3-2
- eqpprice 6-141

- eqpsens 6-144
- eqptimespec 6-147
- eqptree 6-149
- equity binary trees
 - building 3-3
- equity exotic options
 - types 3-26
 - types of 3-22

F

- field 1-9
- fixed lookback options 3-25
- fixed-rate note, defined 2-3
- fixedbybdt 6-153
- fixedbybk 6-156
- fixedbyhjm 6-159
- fixedbyhw 6-162
- fixedbyzero 6-165
- floatbybdt 6-168
- floatbybk 6-171
- floatbyhjm 6-174
- floatbyhw 6-177
- floatbyzero 6-180
- floating lookback option 3-25
- floating-rate note, defined 2-3
- floor, defined 2-3
- floorbybdt 6-183
- floorbybk 6-187
- floorbyhjm 6-190
- floorbyhw 6-193

G

- gamma 2-26
 - defined 4-3

H

- Heath-Jarrow-Morton (HJM) model 2-28
- Heath-Jarrow-Morton tree 2-50

- hedgeopt 6-196
 - purpose 4-2
 - hedgeslf 6-200
 - purpose 4-3
 - hedging
 - considerations 4-2
 - functions 4-2
 - goals 4-2
 - HJM model
 - described 2-3
 - HJM pricing options structure A-2
 - hjmprice 6-204
 - hjmsens 6-208
 - hjmtimespec 6-211
 - hjmtree 6-213
 - input arguments 2-29
 - HJMTree 2-50
 - hjmvolspec 6-215
 - forms of volatility 2-30
 - Hull-White (HW) model 2-28
 - HW model 2-4
 - hwprice 6-219
 - hwsens 6-222
 - hwtimespec 6-225
 - hwtree 6-227
 - hwvolspec 6-230
- I**
- implied trinomial trees
 - building 3-8
 - inconsistent constraints 4-25
 - instadd 6-232
 - creating an instrument 1-4
 - instaddfield 6-235
 - creating new instruments 1-9
 - instasian 6-239
 - instbarrier 6-242
 - instbond 6-244
 - instcap 6-247
 - instcf 6-249
 - instcompound 6-251
 - instdelete 6-254
 - instdisp 6-257
 - instfields 6-259
 - instfind 6-262
 - purpose 1-11
 - syntax 1-11
 - instfixed 6-265
 - instfloat 6-267
 - instfloor 6-269
 - instget 6-271
 - instgetcell 6-276
 - instlength 6-281
 - instlookback 6-282
 - instoptbnd 6-284
 - instoptstock 6-287
 - instrument
 - creating 1-9
 - instrument constructor 1-8
 - instrument index 1-11
 - instselect 6-290
 - purpose 1-11
 - instsetfield 6-293
 - instswap 6-297
 - insttypes 6-300
 - intenvget 6-302
 - purpose 2-21
 - intenvprice 6-304
 - intenvsens 6-306
 - intenvset 6-309
 - purpose 2-19
 - interest rate term structure, defined 2-9
 - inverse discount 2-39
 - isafin 6-314
 - ITT model description 3-2
 - ittprice 6-315
 - ittsens 6-318
 - itttimespec 6-321
 - itttree 6-322

L

- least squares problem 4-20
- lookback option
 - defined 3-24
 - types of 3-24
- lookbackbycrr 6-327
- lookbackbyeqp 6-330
- lookbackbyitt 6-333

M

- mkbush 6-336
- mtree 6-338
- mktrintree 6-339
- mmktbybdt 6-340
- mmktbyhjm 6-342
- model
 - Black-Derman-Toy (BDT) 2-28
 - Black-Karasinski (BK) 2-28
 - Cox-Ross-Rubinstein (CRR) 3-2
 - Equal Probabilities (EQP) 3-2
 - Heath-Jarrow-Morton (HJM) 2-28
 - Hull-White (HJW) 2-28
 - Implied Trinomial Tree (ITT) 3-2
- multifactor volatility models 2-31

O

- object 1-8
- observation time zero 2-55
- optbnbybk 6-349
- optbnbyhw 6-359
- optbndbybdt 6-344
- optbndbyhjm 6-354
- Options argument
 - input to pricing functions 2-50
- optstockbycrr 6-364
- optstockbyeqp 6-367
- optstockbyitt 6-370

P

- per-dollar sensitivities
 - calculating 2-60
 - example 2-27
- portfolio 1-4
 - creation 1-4
 - management 1-8
- portfolio pricing functions
 - equity derivatives 3-26
 - interest-rate based 2-49
- price tree structure 2-56
- Price vector
 - BDT 2-58
 - HJM 2-55
- pricing options
 - default structure A-2
 - structure A-2

R

- rate specification 2-9
- rate2disc 6-373
 - creating inverse discounts 2-40
 - purpose 2-9
- RateSpec
 - creation of 2-30
 - defined 2-9
 - using with HJM 2-34
- ratetimes 6-378
 - purpose 2-9
- rebate 3-24
- recombining trees 2-5
- root node 6-424

S

- sensitivity
 - per-dollar, viewing 2-60
 - types of 2-26
- sensitivity functions 2-58

- short rate 2-4
- specific-instrument pricing functions 2-50
- stochastic model 2-23
- stock structure 3-4 3-9
- stockoptspec 6-382
- StockOptSpec
 - for stock trees 3-12
- stockspec 6-386
- swap, defined 2-3
- swapbybdt 6-388
- swapbybk 6-393
- swapbyhjm 6-398
- swapbyhw 6-404
- swapbyzero 6-409

T

- time2date 6-413
- TimeSpec
 - defined 2-30
 - for stock trees 3-5 3-10
 - using 2-35
- treepath 6-417
- trees
 - binomial 2-5
 - bushy 2-5
 - recombining 2-5
 - trinomial 2-5

- treeshape 6-419
- treeviewer 6-421
 - displaying BDT trees 6-430
 - displaying HJM trees 6-425
 - examining values with 2-61
 - purpose 2-6
 - with recombining trees 2-64
- trinomial trees 2-5
- trintreepath 6-438
- trintreeshape 6-440
- TypeString argument 1-4

U

- under-determined system 4-21

V

- vanilla swaps 2-3
- vega, defined 4-3
- volatility
 - process 2-31
- VolSpec
 - BDT 2-32
 - calling syntax 2-30
 - HJM 2-30
 - using 2-31