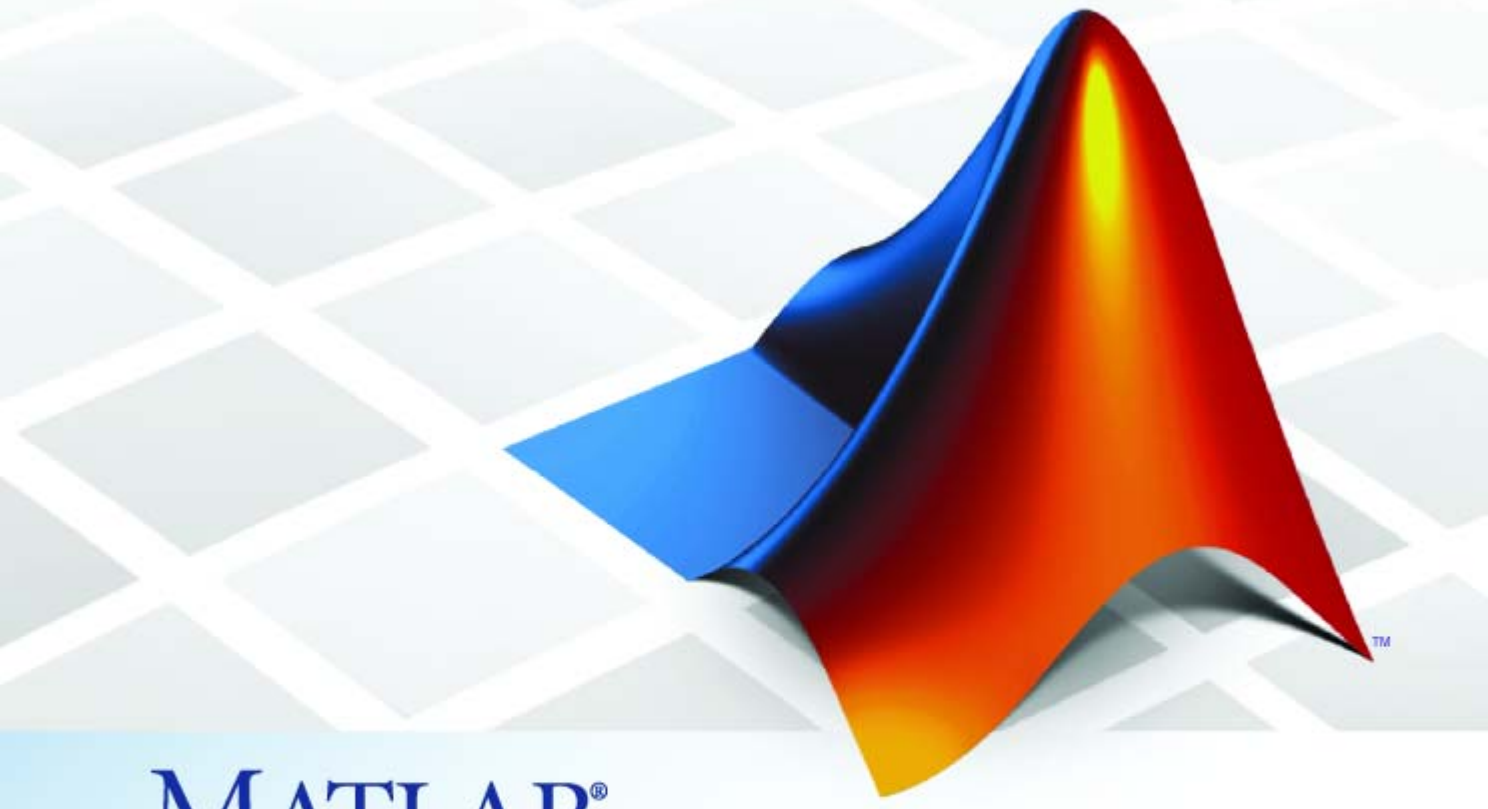
Financial Derivatives Toolbox™ 5
User's Guide



MATLAB®

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support

`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

Financial Derivatives Toolbox™ User's Guide

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See `www.mathworks.com/trademarks` for a list of additional trademarks. Other product or brand names may be 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)
September 2007	Sixth printing	Revised for Version 5.1 (Release 2007b)
March 2008	Online only	Revised for Version 5.2 (Release 2008a)
October 2008	Online only	Revised for Version 5.3 (Release 2008b)
March 2009	Online only	Revised for Version 5.4 (Release 2009a)
September 2009	Online only	Revised for Version 5.5 (Release 2009b)
March 2010	Online only	Revised for Version 5.5.1 (Release 2010a)

Contents

1 Getting Started

2 Interest-Rate Derivatives

3 Equity Derivatives

4 Hedging Portfolios

5 Function Reference

Examples

C

1

Getting Started

Product Overview

In this section...

Introduction

Financial Derivatives Toolbox™ software 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

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

- Bonds
- Bond options (puts and calls)
- Caps
- Fixed-rate notes
- Floating-rate notes
- Floors
- Swaps
- Swaption
- Callable and Puttable bonds

Additionally, the toolbox provides functions to create *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 2-30 or "Computing Prices and Sensitivities Using Interest-Rate Tree Models" on page 2-62 for information.

Equity-Based Derivatives

The toolbox also provides functions to create and manage various equity-based derivatives, including the following:

- 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 Using Trees" on page 3-32.)

Expected Background

In general, this guide assumes experience working with financial derivatives and some familiarity with the underlying models.

In designing Financial Derivatives Toolbox documentation, we assume your title is similar to one of these:

- Analyst, quantitative analyst
- Risk manager
- Portfolio manager
- Fund manager, asset manager
- Financial engineer
- Trader
- Student, professor, or other academic

We also assume your background, education, training, and responsibilities match some aspects of this profile:

- Finance, economics, perhaps accounting
- Engineering, mathematics, physics, other quantitative sciences
- Bachelor's degree minimum; MS or MBA likely; Ph.D. perhaps; CFA
- Comfortable with probability theory, statistics, and algebra
- Understand linear or matrix algebra, calculus, and differential equations
- Previously done traditional programming (C, Fortran, etc.)
- Responsible for instruments or analyses involving large sums of money
- Perhaps new to MATLAB

Portfolio Creation

In this section...

Introduction

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', CFlowAmounts, CFlowDates, Settle, Basis)
```

- Fixed-rate note instrument

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

- Floating-rate note instrument

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

- Cap instrument

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

- Floor instrument

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

- Swap instrument

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

- Swaption instrument

```
InstSet = instadd('Swaption', OptSpec, Strike, ExerciseDates, Spread, ...
Settle, Maturity, AmericanOpt, SwapReset, Basis, Principal)
```

- Bond with embedded option instrument

```
InstSet = instadd('OptEmBond', CouponRate, Settle, Maturity, OptSpec, Strike, ...
ExerciseDates, 'AmericanOpt', AmericanOpt, 'Period', Period,'Basis', Basis, ...
'EndMonthRule', EndMonthRule,'Face',Face,'IssueDate', IssueDate, 'FirstCouponDate', ...
FirstCouponDate, 'LastCouponDate', LastCouponDate,'StartDate', StartDate)
```

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 EndMonthRule IssueDate ... Face
3     Bond 0.04       08-Feb-2000    15-Jan-2003  2      0     1            NaN       ... 100
```

Portfolio Management

In this section...

Instrument Constructors

The toolbox provides constructors for the most common financial instruments. 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`
Swaption	`instswaption`

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

- 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. You have the ability to add additional parameters. These additional fields are 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 kind of instrument or to add new properties to the instruments in an existing instrument collection.

To create a kind of instrument with `instaddfield`, you must specify three arguments:

- `Type`
- `FieldName`
- `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 argument is `ClassList`. `ClassList` specifies the data types of the contents of each unique field for the new instrument.

Use either syntax to create a 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 software does not directly support this instrument, you must create it using the function `instaddfield`. Use these parameters to create instruments:

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

Enter the data into MATLAB® software:

```
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 Jul-15-2000 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 Principal
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 software 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 categorized by type, each type can have different data fields. The stored data field is a row vector or string for each instrument.

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

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

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

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

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

```
load deriv.mat
instdisp(HJMInstSet)

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

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

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

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

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

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

Index Type LegRate   Settle     Maturity     LegReset Basis Principal LegType  Name   Quantity

8     Swap [0.06 20] 01-Jan-2000  01-Jan-2003  [1  1]   NaN   NaN     [NaN]  6%/20BP Swap  10
```

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

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

Mat2003 =

     1
     4
     5
     8
```

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

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

CapFloor =

     6
     7
```

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

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

Pos50 =

     2
     3
```

instselect

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

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

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

Mat2003 =

     1
     4
     5
     8
```

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

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

instdisp(Select2003)
```

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

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

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

Index Type LegRate   Settle     Maturity  LegReset Basis Principal LegType Name Quantity
4     Swap [0.06 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 3 instrument types: `Option`, `Futures`, and `TBill`. Use `instselect` to make a new instrument set containing only options struck at `95`. In other words, select all instruments containing the field `Strike` *and* with the data value for that field equal to `95`.

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

instdisp(InstSet)

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

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

2

Interest-Rate Derivatives

Understanding Interest-Rate Derivative Instruments

In this section...

Introduction

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

The toolbox provides a set of functions that perform computations on portfolios containing the following interest-rate based financial instruments:

- Bond
- Bond options
- Bond with embedded options
- Fixed-rate note
- Floating-rate note
- Cap

- Floor
- Swap
- Swaption

Additionally, Financial Derivatives Toolbox software lets you create and price arbitrary cash flow instruments based on zero-coupon bonds or on any supported interest-rate model. For more information, see "Interest-Rate Modeling" on page 2-10.

Bond

A *bond* is a long-term debt security with a preset interest-rate and maturity. At maturity you must pay the principal and interest.

The price or value of a bond is determined by discounting the expected cash flows of the bond to the present, using the appropriate discount rate. The following equation represents the relationship of the expected cash flows and discount rate:

$$B_0 = \frac{C}{2}\left[\frac{1-\left(1+\frac{r}{2}\right)^{-2t}}{\frac{r}{2}}\right] + \frac{F}{\left(1+\frac{r}{2}\right)^{2t}}$$

where:

B_0 is the bond value.

C is the annual coupon payment.

F is the face value of the bond.

r is the required return on the bond.

t is the number of years remaining until maturity.

Financial Derivatives Toolbox supports the following for pricing and specifying a bond.

Function	Purpose
bondbybdt	Price a bond using a BDT interest-rate tree.
bondbyhw	Price a bond using an HW interest-rate tree.
bondbybk	Price a bond using a BK interest-rate tree.
bondbyhjm	Price a bond using an HJM interest-rate tree.
bondbyzero	Price a bond using a set of zero curves.
instbond	Construct a bond instrument.

Bond Options

Financial Derivatives Toolbox software supports three types of put and call options on bonds:

- American option: An option that you exercise any time until its expiration date.
- European option: An option that you exercise only on its expiration date.
- Bermuda option: A Bermuda option resembles a hybrid of American and European options. You can exercise it on predetermined dates only, usually monthly.

Financial Derivatives Toolbox supports the following for pricing and specifying a bond option.

Function	Purpose
optbndbybdt	Price a bond option price using a BDT interest-rate tree.
optbndbyhw	Price a bond option price using an HW interest-rate tree.
optbndbybk	Price a bond option price using a BK interest-rate tree.
optbndbyhjm	Price a bond option price using an HJM interest-rate tree.
instoptbnd	Construct a bond option instrument.

Bond with Embedded Options

A bond with embedded options allows the issuer to buy back or redeem the bond at a predetermined price at specified future dates. Financial Derivatives Toolbox software supports American, European, and Bermuda callable and puttable bonds.

The pricing for a bond with embedded options is as follows:

- For a callable bond: *PriceCallableBond = BondPrice - BondCallOption*
- For a puttable bond: *PricePuttableBond = PriceBond + PricePutOption*

Financial Derivatives Toolbox supports the following for pricing and specifying a bond with embedded options.

Function	Purpose
`optembndbybdt`	Price a bond with embedded options using a BDT interest-rate tree.
`optembndbyhw`	Price a bond with embedded options using an HW interest rate tree.
`optembndbybk`	Price a bond with embedded options using a BK interest-rate tree.
`optembndbyhjm`	Price a bond with embedded options using an HJM interest-rate tree.
`instoptembnd`	Construct a bond-with-embedded-options instrument.

Fixed-Rate Note

A *fixed-rate note* is a long-term debt security with a preset interest rate and. At maturity the interest must be paid. The principal may or may not be paid at maturity. In Financial Derivatives Toolbox software, the principal is always paid at maturity.

Financial Derivatives Toolbox supports the following for pricing and specifying a fixed-rate note.

Function	Purpose
fixedbybdt	Price a fixed-rate note using a BDT interest-rate tree.
fixedbyhw	Price a fixed-rate note using an HW interest-rate tree.
fixedbybk	Price a fixed-rate note using a BK interest-rate tree.
fixedbyhjm	Price a fixed-rate note using an HJM interest-rate tree.
fixedbyzero	Price a fixed-rate note using a set of zero curves.
instfixed	Construct a fixed-rate instrument.

Floating-Rate Note

A *floating-rate note* is a security like a bond, but the interest rate of the note is reset periodically, relative to a reference index rate, to reflect fluctuations in market interest rates.

Financial Derivatives Toolbox supports the following for pricing and specifying a floating-rate note.

Function	Purpose
floatbybdt	Price a floating-rate note using a BDT interest-rate tree.
floatbyhw	Price a floating-rate note using an HW interest-rate tree.
floatbybk	Price a floating-rate note using a BK interest-rate tree.
floatbyhjm	Price a floating-rate note using an HJM interest-rate tree.
floatbyzero	Price a floating-rate note using a set of zero curves.
instfloat	Construct a floating-rate note instrument.

Cap

A *cap* is 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. The payoff for a cap is:

$$\max(CurrentRate - CapRate, 0)$$

Financial Derivatives Toolbox supports the following for pricing and specifying a cap instrument.

Function	Purpose
`capbybdt`	Price a cap instrument using a BDT interest-rate tree.
`capbyhw`	Price a cap instrument using an HW interest-rate tree.
`capbybk`	Price a cap instrument using a BK interest-rate tree.
`capbyhjm`	Price a cap instrument using an HJM interest-rate tree.
`capbyblk`	Price a cap instrument using the Black option pricing model.
`instcap`	Construct a cap instrument.

Floor

A *floor* is a contract that includes a guarantee setting the minimum interest rate to be received by the holder, based on an otherwise floating interest rate. The payoff for a floor is:

$$\max(FloorRate - CurrentRate, 0)$$

Financial Derivatives Toolbox supports the following for pricing and specifying a floor instrument.

Function	Purpose
floorbybdt	Price a floor instrument using a BDT interest-rate tree.
floorbyhw	Price a floor instrument using an HW interest-rate tree.
floorbybk	Price a floor instrument using a BK interest-rate tree.
floorbyhjm	Price a floor instrument using an HJM interest-rate tree.
instfloor	Construct a floor instrument.

Swap

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

Financial Derivatives Toolbox supports the following for pricing and specifying a swap instrument.

Function	Purpose
swapbybdt	Price a swap instrument using a BDT interest-rate tree.
swapbyhw	Price a swap instrument using an HW interest-rate tree.
swapbybk	Price a swap instrument using a BK interest-rate tree.
swapbyhjm	Price a swap instrument using an HJM interest-rate tree.
swapbyzero	Price a swap instrument using a set of zero curves.
instswap	Construct a swap instrument.

Swaption

A *swaption* is an option to enter into an interest-rate swap contract. A call swaption allows the option buyer to enter into an interest-rate swap where the buyer of the option pays the fixed-rate and receives the floating-rate. A put swaption allows the option buyer to enter into an interest-rate swap where the buyer of the option receives the fixed-rate and pays the floating-rate.

Financial Derivatives Toolbox supports the following for pricing and specifying a swaption instrument.

Function	Purpose
`swaptionbybdt`	Price a swaption instrument using a BDT interest-rate tree.
`swaptionbyhw`	Price a swaption instrument using an HW interest-rate tree.
`swaptionbybk`	Price a swaption instrument using a BK interest-rate tree.
`swaptionbyhjm`	Price a swaption instrument using an HJM interest-rate tree.
`instswaption`	Construct a swaption instrument.

Overview of Interest-Rate Models

In this section...

Interest-Rate Modeling

Financial Derivatives Toolbox software 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 1-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 software 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 2-30 for a discussion of price and sensitivity based on portfolios of zero-coupon bonds
- "Computing Prices and Sensitivities Using Interest-Rate Tree Models" on page 2-62 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 software 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 2-57 explains the few noteworthy differences among the various formats.

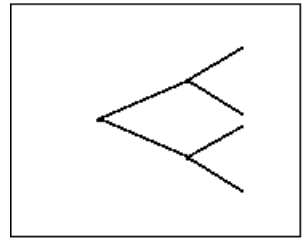
If you need more detailed information about functions that use the HW and BK tree structures, see Chapter 5, "Function Reference", which provides extensive reference information for all functions that compose this toolbox.

Rate and Price Trees

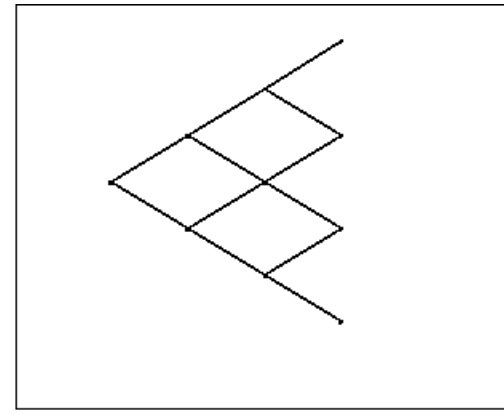
The interest-rate or price trees supported in this toolbox can be either *binomial* (two branches per node) or *trinomial* (3 branches per node). Typically, binomial trees assume that underlying interest rates or prices can only either increase or decrease at each node. Trinomial trees allow for a more 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.

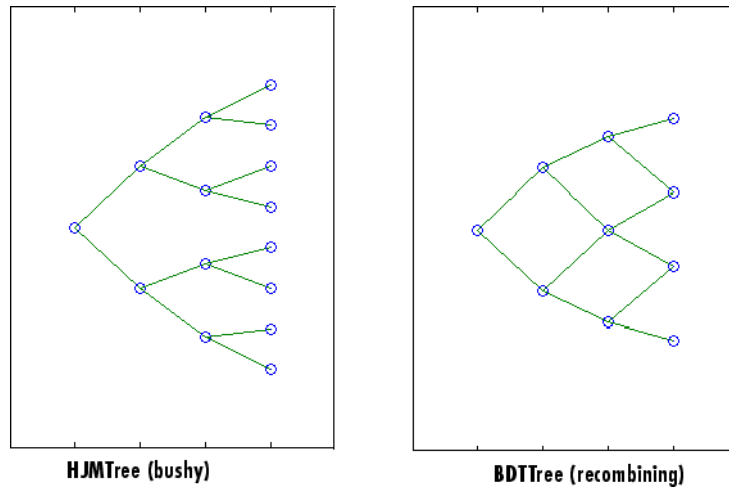
Viewing Rate or Price Movement with This Toolbox

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

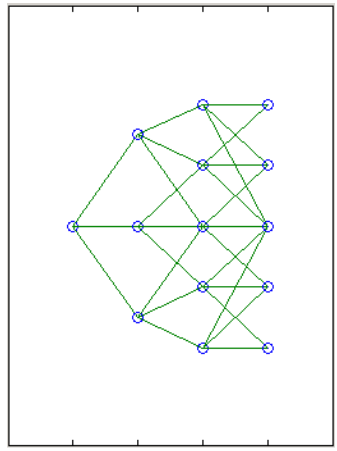
- `HJMTree` — A bushy binomial tree
- `BDTTree` — A recombining binomial tree

- `HWTree` 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.



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


HWTree and BKTree (recombining)

Understanding the Interest-Rate Term Structure

In this section...

Introduction

The *interest-rate term structure* represents the evolution of interest rates through time. In MATLAB software, the interest-rate environment is encapsulated in a structure called `RateSpec` (*rate specification*). This structure holds all information required to completely identify the evolution of interest rates. Several functions included in Financial Derivatives Toolbox software are dedicated to the creating and managing 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 Versus Discount Factors

Discount factors are coefficients commonly used to find the current 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
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 rates table (see "Calculating Discount Factors from Rates" on page 2-15) 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 0s. Also, since the value of `Compounding` is 2, the rates are compounded semiannually, which sets the units of periodic discount to 6 months. The vector `EndDates` is composed of dates separated by intervals of 6 months from the valuation date. This explains why the `EndTimes` vector is a progression of integers from 1 to 5.

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 0, 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 in “Calculating Discount Factors from Rates” on page 2-15.

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

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 like the previous one. 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
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 6-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 software 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:

- "Creating or Modifying (intenvset)" on page 2-25 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 2-27 for a discussion of how to extract specific properties from a `RateSpec`

Creating or Modifying (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 `RateSpec`.

Use `intenvset` to create or modify an interest-rate's term structure`RateSpec`. If the first argument to `intenvset` is a previously created `RateSpec`, the function modifies the existing rate specification and returns a new one; otherwise, `intenvset` creates a `RateSpec`.

When using `RateSpec` to specify the rate term structure to price instruments based on yields (zero coupon rates) or forward rates, specify zero rates or forward rates as the input argument. However, the `RateSpec` structure is not limited or specific to this problem domain. `RateSpec` is an encapsulation of rates-times relationships; `intenvset` acts as either a constructor or a modifier, and `intenvget` as an accessor. The interest rate models supported by the Financial Derivatives Toolbox software work either with zero coupon rates or forward rates.

The other `intenvset` arguments are property-value pairs, indicating the new value for these properties. The properties that can be specified or modified are:

- `Basis`
- `Compounding`
- `Disc`
- `EndDates`
- `EndMonthRule`

- `Rates`
- `StartDates`
- `ValuationDate`

To learn about the properties `EndMonthRule` and `Basis`, type `help ftbEndMonthRule` and `help ftbBasis` or see the Financial Toolbox documentation.

Consider again the original table of interest rates (see "Calculating Discount Factors from Rates" on page 2-15).

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, and the `ValuationDate`, `intenvset` has all the information required to calculate `StartTimes` and `EndTimes`. Hence, these two properties are read-only.

Obtaining Specific Properties (intenvget)

The complementary function to `intenvset` is `intenvget`, which gets function specific properties from the interest-rate term structure. Its syntax 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`.

Caution Although you can directly access these fields within the structure instead of using `intenvget`, it is advised not to do so. 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 6 months is represented with a value of 0.5, and a period of 1 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 0. On the other hand, the values in the `EndDates` vector are dates separated by 6-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

In this section...

Introduction

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 software also contains functions that use the Heath-Jarrow-Morton (HJM) and Black-Derman-Toy (BDT) models to compute prices and sensitivities for financial instruments. These models support computations involving options and interest-rate floors and caps. See "Computing Prices and Sensitivities Using Interest-Rate Tree Models"

on page 2-62 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

`bondbyzero`:	Price a bond by a set of zero curves
`fixedbyzero`:	Price a fixed-rate note by a set of zero curves
`floatbyzero`:	Price a floating-rate note by a set of zero curves
`swapbyzero`:	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 `intenvset`, 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 software.

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 and 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 Tree Models

In this section...

Introduction

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

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 2-57.

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 2-38.)
- The interest-rate term structure `RateSpec`. (See "Specifying the Interest-Rate Term Structure (RateSpec)" on page 2-41.)
- The tree time layout `TimeSpec`. (See "Specifying the Time Structure (TimeSpec)" on page 2-47.)

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 2-75 for information about `treeviewer`.

Calling Sequence

The calling syntax for `hjmtree` is

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

Similarly, the calling syntax for `bdttree` is

```
BDTTree = 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 2-24.)
- `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 3) volatility models while BDT (also, BK and HW) supports only a single volatility factor, the `hjmvolspec` and `bdtvolspec` functions require different inputs and generate slightly different outputs. For examples, see "Creating an HJM Volatility Model" on page 2-38. For BDT examples see "Creating a BDT Volatility Model" on page 2-40.

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 3 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 following 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 2-24 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 following 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
```

Calibrating the Hull-White Model Using Market Data

The pricing of interest rate derivative securities relies on models that describe the underlying process. These interest rate models depend on one or more parameters that you must determine by matching the model predictions to the existing data available in the market. In the Hull-White model, there are two parameters related to the short rate process: mean reversion and volatility. Determining these parameters, such that the model is able to reproduce as close as possible the prices of caps or floors observed in the market, is called calibration. The calibration routines find the parameters that minimize the difference between the model price predictions and the market prices for caps and floors.

In the case of the Hull-White model, the minimization is two dimensional with respect to mean reversion (α) and volatility (σ). That is, calibrating the Hull-White model minimizes the difference between the model prices and market prices for caps and floors:

$$\frac{(\text{ModelPrice}(\alpha\ ,\ \sigma) - \text{MarketPrice})}{(\text{MarketPrice})}$$

Hull-White Model Calibration Example

You can use `hwcalbycap` and `hwcalbyfloor` to determine the volatility parameters, mean reversion (α) and volatility (σ), such that the model fits the prices of actively traded instruments for caps and floors as closely as possible. Consider the following caplet market information:

```
MarketMat = {'21-Mar-2008';
             '21-Jun-2008';
```

```
                '21-Sep-2008';
                '21-Dec-2008';
                '21-Mar-2009';
                '21-Jun-2009';
                '21-Sep-2009';
                '21-Dec-2009';
                '21-Mar-2010';
                '21-Jun-2010';
                '21-Sep-2010';
                '21-Dec-2010';
                '21-Mar-2011'};

MarketStrike = [0.0590; 0.0690; 0.0790];

MarketVol = [0.1533 0.1731 0.1727 0.1752 0.1809 0.1800 0.1805 0.1802 0.1735 0.1757 ...
             0.1755 0.1755 0.1726;
             0.1525 0.1725 0.1725 0.1750 0.1800 0.1800 0.1800 0.1800 0.1725 0.1750 ...
             0.1750 0.1750 0.1725;
             0.1526 0.1730 0.1726 0.1747 0.1808 0.1792 0.1797 0.1794 0.1733 0.1751 ...
             0.1750 0.1745 0.1719];
```

1 Create `RateSpec` using the following data:

```
Rates= [0.0627;
        0.0657;
        0.0691;
        0.0717;
        0.0739;
        0.0755;
        0.0765;
        0.0772;
        0.0779;
        0.0783;
        0.0786;
        0.0789;
        0.0792;
        0.0793];

ValuationDate = '21-Jan-2008';
EndDates = {'21-Mar-2008';'21-Jun-2008';'21-Sep-2008';'21-Dec-2008';...
```

```
            '21-Mar-2009';'21-Jun-2009';'21-Sep-2009';'21-Dec-2009';....
            '21-Mar-2010';'21-Jun-2010';'21-Sep-2010';'21-Dec-2010';....
            '21-Mar-2011';'21-Jun-2011'};
Compounding = 4;

RateSpec = intenvset('ValuationDate', ValuationDate, ...
'StartDates', ValuationDate, 'EndDates', EndDates, ...
'Rates', Rates, 'Compounding', Compounding, 'Basis', 0);
```

2 Call the calibration routine to find the values for the volatility parameters α and σ:

```
Settle = ' Jan-21-2008';
Maturity = 'Mar-21-2011';
Strike = 0.0690;
Reset = 4;
Principal = 1000;
Basis = 0;
o=optimset('TolFun',100*eps);

[Alpha, Sigma] = hwcalbycap(RateSpec, MarketStrike, MarketMat, MarketVol,...
Strike, Settle, Maturity, 'Reset', Reset, 'Principal', Principal, 'Basis',...
Basis, 'OptimOptions', o)

Warning: LSQNONLIN did not converge to an optimal solution. It exited with
exitflag = 2.
LSQNONLIN Diagnostic Message: '
Local minimum possible.

lsqnonlin stopped because the size of the current step is less than
the default value of the step size tolerance.

> In hwcalbycapfloor at 85
  In hwcalbycap at 77

Alpha =

  1.0000e-006
```

```
Sigma =

    0.0127
```

The warning above indicates that the conversion was not optimal. The search algorithm used by the Optimization Toolbox™ function `lsqnonlin` could not find a solution that complies with all the constraints used by the function. To discern whether the solution is acceptable, you must look at the results of the optimization by specifying a third output (`OptimOut`) for `hwcalbycap`.

```
[Alpha, Sigma, OptimOut] = hwcalbycap(RateSpec, MarketStrike, MarketMat,...
MarketVol, Strike, Settle, Maturity, 'Reset', Reset, 'Principal', Principal,...
'Basis', Basis, 'OptimOptions', o);
```

The `OptimOut.residual` field of the `OptimOut` structure is the optimization residual. This value contains the difference between the Black caplets and those calculated during the optimization. You can use the `OptimOut.residual` value to calculate the percentual difference (error) compared to Black Caplet prices and then decide whether the residual is acceptable. There will almost always be some residual, so you must decide if parametrizing the market with a single value of alpha and sigma is acceptable.

3 You can price the caplets using the market data and Black's formula to obtain the reference caplet values:

```
MarketMatNum = datenum(MarketMat);
[Mats,Strikes] = meshgrid(MarketMatNum, MarketStrike);
FlatVol = interp2(MarketMatNum, MarketStrike, MarketVol, datenum(Maturity),...
Strike, 'spline');
[CapPrice, Caplets] = capbyblk(RateSpec, Strike, Settle, Maturity, FlatVol,...
'Reset', Reset, 'Basis', Basis, 'Principal', Principal);
Caplets = Caplets(2:end)';
```

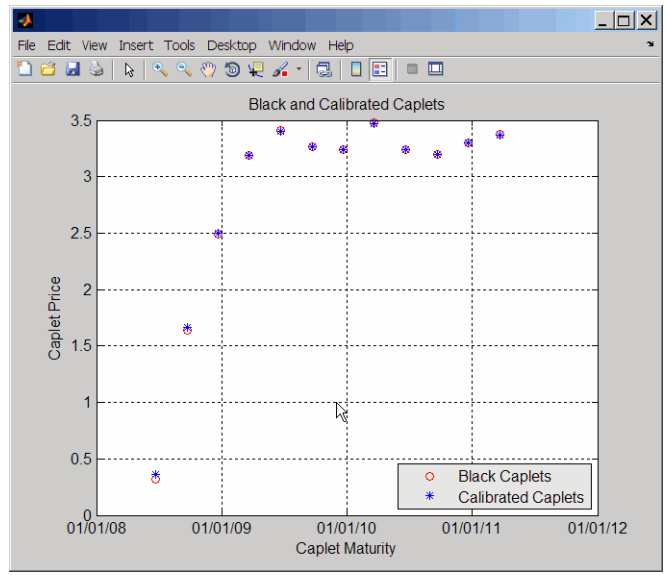
4 You can compare the optimized values and the Black values and display graphically:

```
OptimCaplets = Caplets+OptimOut.residual;
disp([Caplets OptimCaplets])
plot(MarketMatNum(2:end), Caplets, 'or', MarketMatNum(2:end), OptimCaplets, '*b');
```

```
datetick('x', 2)
xlabel('Caplet Maturity');
ylabel('Caplet Price');
title('Black and Calibrated Caplets');
h = legend('Black Caplets', 'Calibrated Caplets');
set(h, 'color', [0.9 0.9 0.9]);
set(h, 'Location', 'SouthEast');
set(gcf, 'NumberTitle', 'off')
grid on
```

```
    0.3216    0.3643
    1.6363    1.6612
    2.4872    2.4983
    3.1912    3.1883
    3.4121    3.4051
    3.2698    3.2653
    3.2400    3.2379
    3.4819    3.4699
    3.2437    3.2426
    3.1968    3.1977
    3.3011    3.2980
    3.3771    3.3684
```

5 To visualize this, consider the following comparison:



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 following 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 following 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(BDTVolSpec, 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 software 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 required 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
  ITTTree           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 software 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 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 software uses *inverse discount* notation for forward rates in the tree. An inverse discount represents a factor by which the current value of an asset is multiplied to find its future value. In general, these forward factors are reciprocals of the discount factors.

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

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

ans =

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

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

```
Disc = rate2disc(HJMTree.TimeSpec.Compounding,...
HJMTree.RateSpec.Rates, HJMTree.RateSpec.EndTimes,...
HJMTree.RateSpec.StartTimes);
FRates = 1./Disc

FRates =
    1.0356
    1.0468
    1.0523
    1.0563
```

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

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

```
HJMTree.VolSpec

ans =

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

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

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

ans =

    1.0364
    1.0420
    1.0461
```

Now examine the corresponding down branch.

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

ans =

    1.0574
    1.0631
    1.0672
```

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

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

ans =
```

```
    1.0317    1.0526
    1.0358    1.0568
```

Next examine the corresponding down states.

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

ans =

    1.0526    1.0738
    1.0568    1.0781
```

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

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

FRates =

    1.0356
    1.0364
    1.0526
    1.0674
```

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

Isolating the same node using direct indexing obtains

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

```
ans =

    1.0674
```

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

You can use these techniques with any type of tree generated with Financial Derivatives Toolbox software, 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 software 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 are `VolSpec`, `TimeSpec`, and `RateSpec`, which contain the volatility, time structure, and rate 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, `tObs = 1`. This node contains a total of two states, one representing the branch going up (`1.0979`) and the other representing the branch going down (`1.1432`).

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

```
BDTTree.FwdTree{3}

ans =

    1.0976    1.1377    1.1942
```

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

```
BDTTree.FwdTree{4}
```

```
ans =

    1.0872    1.1183    1.1606    1.2179
```

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

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(BDTTree.FwdTree, [1 2 2])

FRates =

    1.1000
    1.0979
    1.1377
    1.1606
```

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

HW and BK Tree Structures

The HW and BK tree structures are 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 1 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 1 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
```

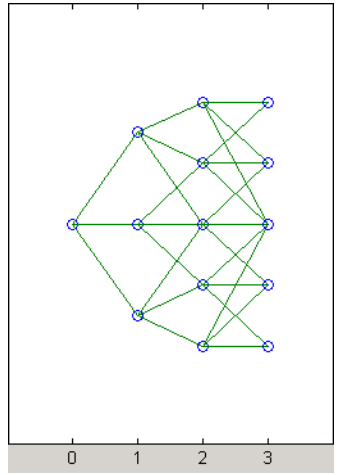
This indicates that the middle branch of the root node connects to the second (from the top) node of the next level, as expected. If you subtract 1 from this value, you obtain 1, which tells you that the up branch goes to the top node. If you add 1, you obtain 3, which points to the last node of the 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 Tree Models

In this section...

Introduction

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 `bdtprice` 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
- Swaptions

For example, the syntax for calling `hjmprice` is:

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

Similarly, the calling syntax for `bdtprice` is:

```
[Price, PriceTree] = bdtprice(BDTTree, 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`. `BDTTree` is the Black-Derman-Toy tree sampling of an interest-rate process, created using `bdttree`. See "Building a Tree of Forward Rates" on page 2-36 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`, `swapbyhjm`, and `swaptionbyhjm`. A similarly named set of functions exists for BDT models. For a list of these, see "Black-Derman-Toy Trees" on page 5-4.

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 required 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
```

```
ITTTree          1x1              8862  struct
ZeroInstSet      1x1             10282  struct
ZeroRateSpec     1x1              1580  struct
```

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

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

```
instdisp(BDTInstSet)
```

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


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

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

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

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

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

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

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

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

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

Price =

   95.5030
   93.9079
    1.7657
   95.5030
  100.4865
    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(HJMTree, 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(BDTTree, BDTInstSet)

PriceTree.PTree{1}

ans =

   95.5030
   93.9079
    1.7657
   95.5030
  100.4865
    1.4863
    0.0245
    7.4222
```

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 `hjmsens` and `bdtsens` 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; `BDTTree` 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 `HJMInstSet`. 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] = bdtsens(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
```

Interest-Rate Derivatives Using Closed Form Solutions

Pricing Caps and Floors Using the Black Option Model

Caps and floors are contracts that allow the holder to be protected if interest rates rise or decrease. The Black model uses a forward price as an underlier in place of a spot price. The assumption is that the forward price at maturity of the option is log-normally distributed.

Closed-form solutions for pricing caps and floors using the Black model support the following tasks:

Task	Function
Price the interest rate caps using the Black option pricing model.	`capbyblk`
Price the interest rate floors using the Black option pricing model.	`floorbyblk`

Graphical Representation of Trees

In this section...

Introduction

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
```

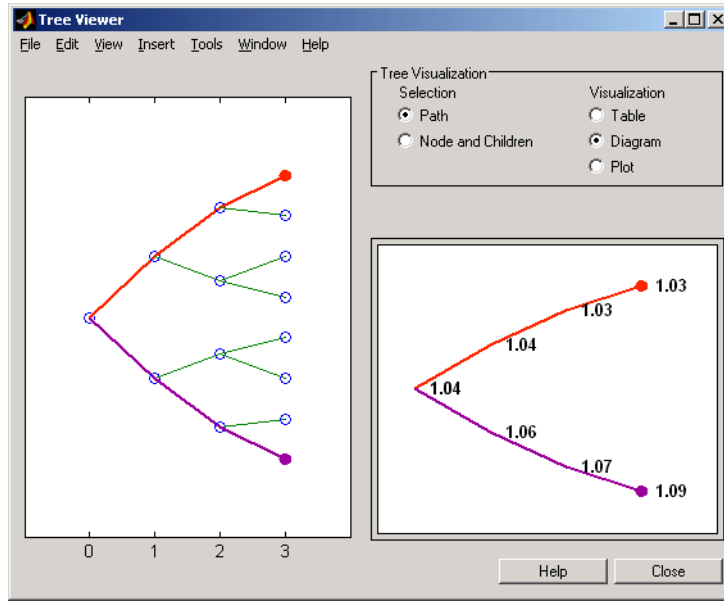
Note `treeviewer` price tree diagrams follow the convention that increasing prices appear on the upper branch of a tree and, consequently, decreasing prices appear on the lower branch. Conversely, for interest rate displays, *decreasing* interest rates appear on the upper branch (prices are rising) and *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 2-75. For information on using `treeviewer` to observe the movement of prices, see "Observing Instrument Prices" on page 2-79.

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`.

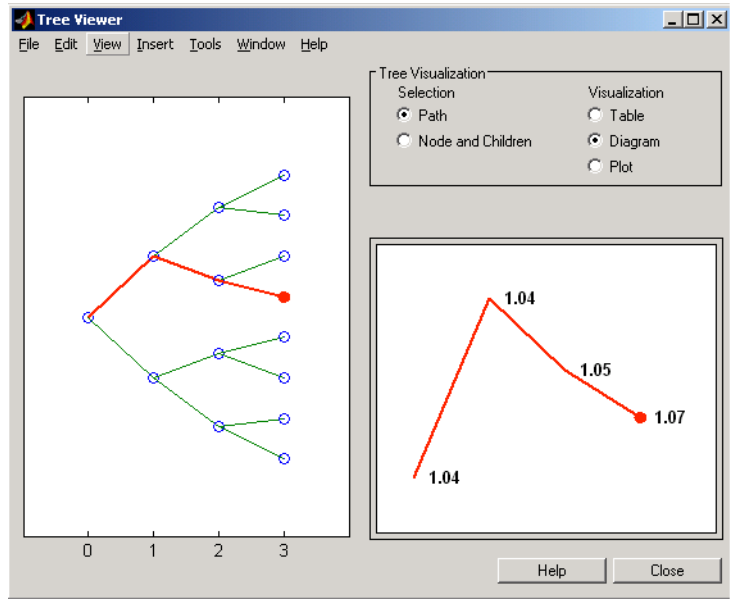
```
treeviewer(HJMTree)
```



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.

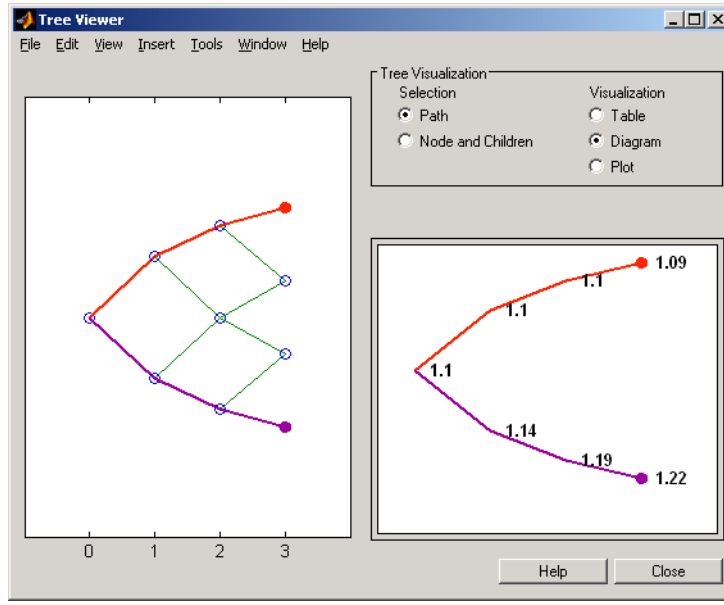
```
FRates = bushpath(HJMTree.FwdTree, [1 2 2])

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)
```

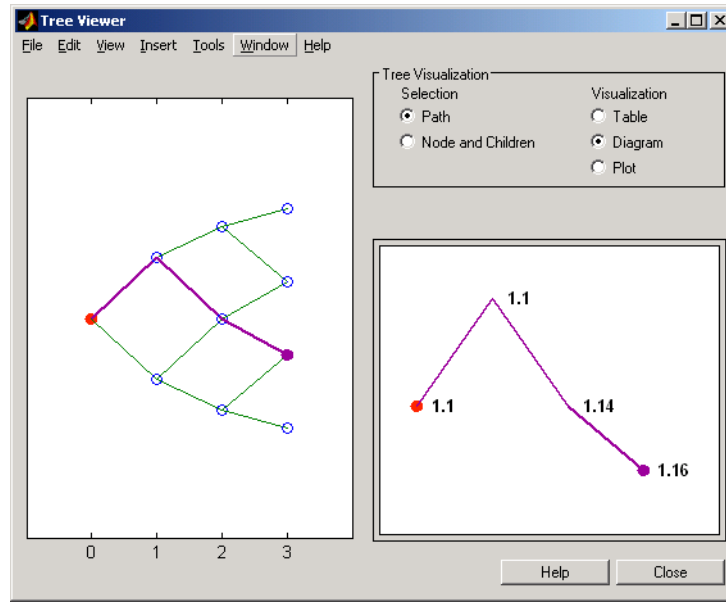


Note When using `treeviewer` with recombining trees, such as BDT, BK, and HW, you must click each node in succession from the beginning to the end. Because these trees can recombine, `treeviewer` is unable to complete the path automatically.

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.

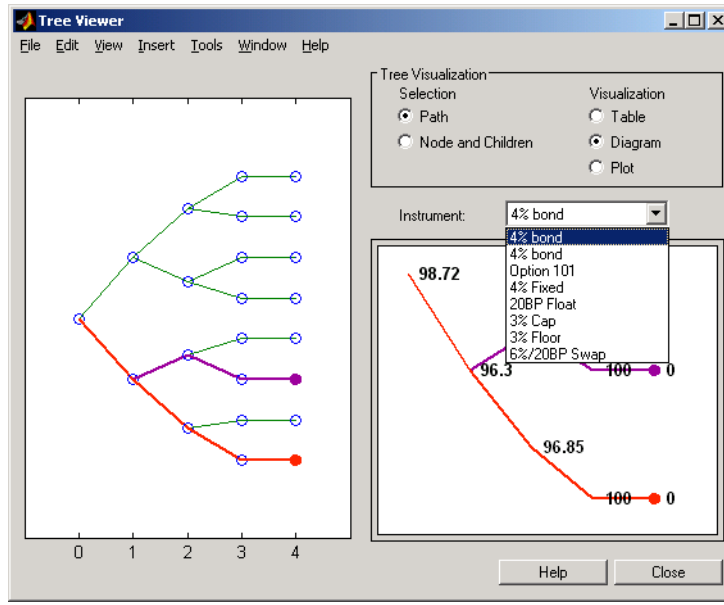


Observing Instrument Prices

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:

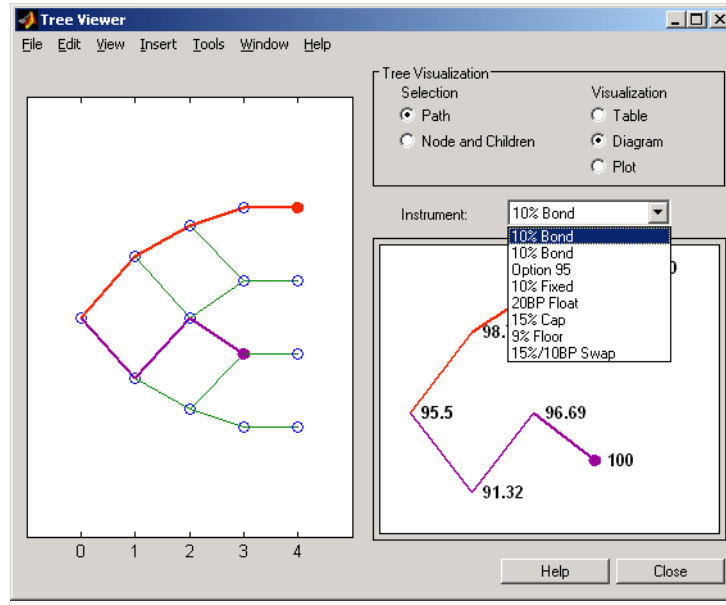
```
load deriv.mat
[Price, PriceTree] = hjmprice(HJMTree, HJMInstSet);
treeviewer(PriceTree, HJMInstSet)
```

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



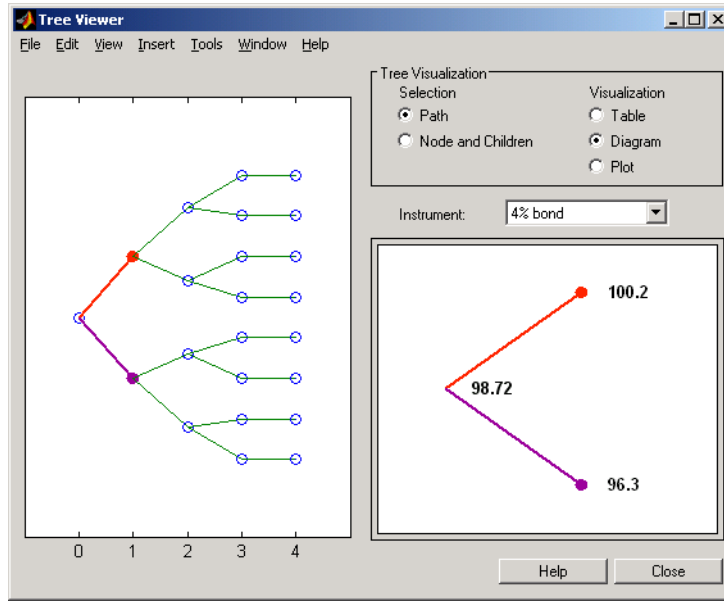
You can use an analogous process to view instrument prices based on the BDT interest rate tree included in `deriv.mat`.

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

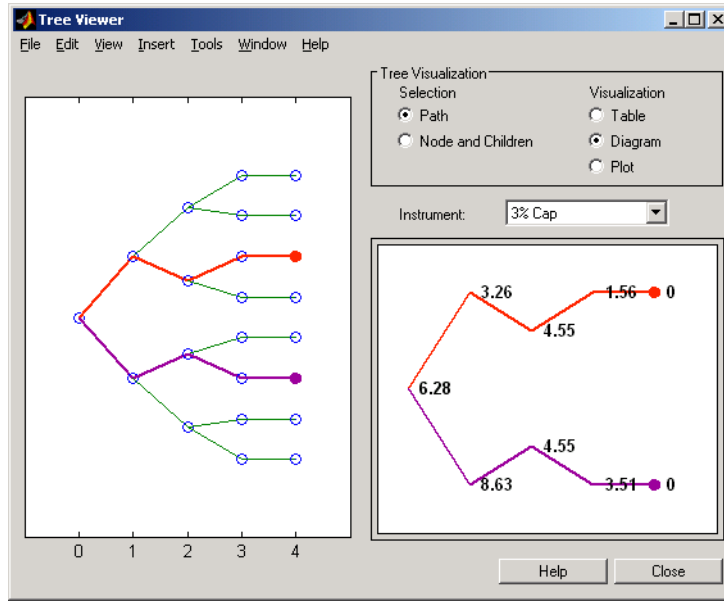


Valuation Date Prices

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.



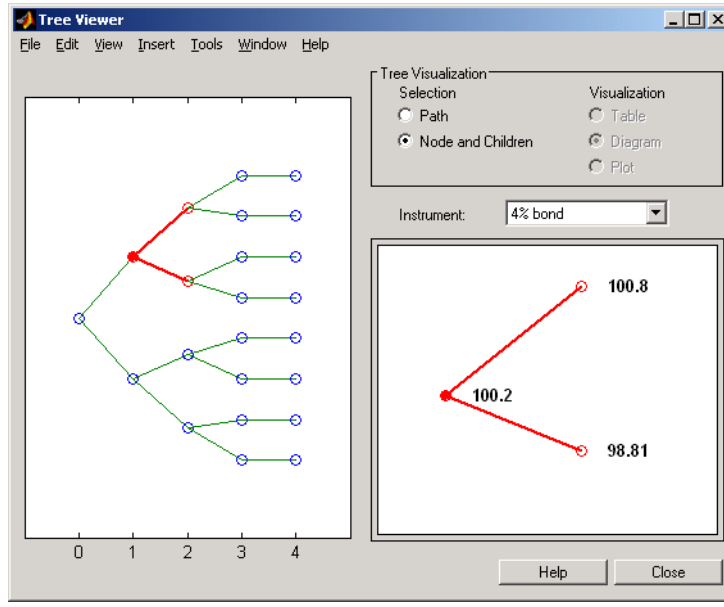
Additional Observation Times

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

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

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

ans =

  100.1563
   99.7309
    0.1007
  100.1563
  100.3782
    3.2594
    0.1007
    3.5597
```

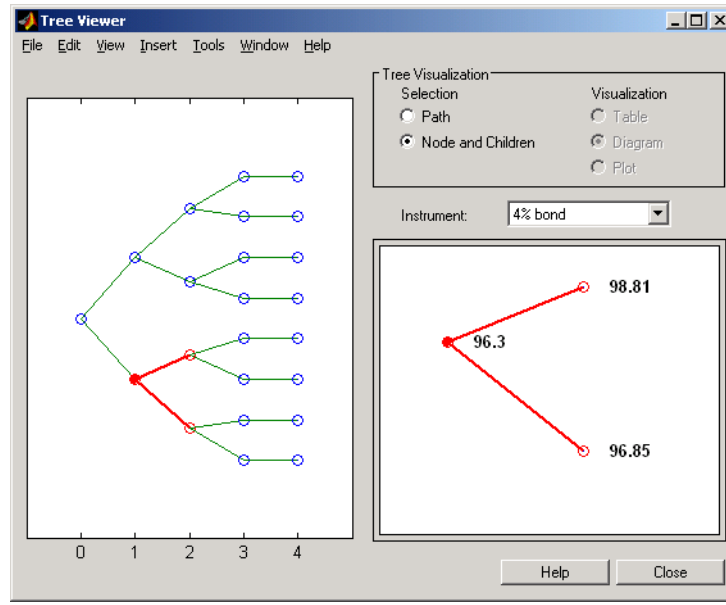
As before, you can use `treeviewer`, this time to examine the price for the 4% bond on the up branch. `treeviewer` displays a price of 100.2 for the first node of the up branch, as expected.



Now examine the corresponding down branch.

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

ans =

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

Use `treeviewer` once again, now to observe the price of the 4% bond on the down branch. The displayed price of 96.3 conforms to the price obtained from direct access of the `PriceTree` structure. You may continue this process as far along the price tree as you want.



3

Equity Derivatives

Understanding Equity Trees

In this section...

Introduction

Financial Derivatives Toolbox software 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 2-11.

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 many 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 2-24 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 = crrtimespec(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 required 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:

```
ITTTree = 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 2-24 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: 'ITTTimeSpec'
```

```
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 ITTTree, enter:

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

ITTTree =

          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 software 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 required 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
ITTInstSet       1x1            12438  struct
ITTTree          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 1 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 ITTTree

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

```
TTTree =

          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 itttree. 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, `ITTTree.STree{4}(1)` represents the top element of the fourth level of the tree, and `ITTTree.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 ITTTree

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(ITTTree, [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

In this section...

Introduction

Financial Derivatives Toolbox software supports eight types of equity exotic options. 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 option 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.

The following functions support Asian options.

Function	Purpose
`asianbycrr`	Price Asian options from a CRR binomial tree.
`asianbyeqp`	Price Asian options from an EQP binomial tree.
`asianbyitt`	Price Asian options using an implied trinomial tree (ITT).
`instasian`	Construct an Asian option.

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.

The following functions support barrier options.

Function	Purpose
`barrierbycrr`	Price barrier options from a CRR binomial tree.
`barrierbyeqp`	Price barrier options from an EQP binomial tree.
`barrierbyitt`	Price barrier options using an implied trinomial tree (ITT).
`instbarrier`	Construct a barrier option.

Basket Option

A *basket* option is an option on a portfolio of several underlying equity assets. Payout for a basket option depends on the cumulative performance of the collection of the individual assets. A basket option tends to be cheaper than the corresponding portfolio of plain vanilla options for these reasons:

- If the basket components correlate negatively, movements in the value of one component neutralize opposite movements of another component. Unless all the components correlate perfectly, the basket option is cheaper than a series of individual options on each of the assets in the basket.
- A basket option minimizes transaction costs because an investor has to purchase only one option instead of several individual options.

The payoff for a basket option is as follows:

- For a call: $\max(\sum Wi * Si - K; 0)$
- For a put: $\max(\sum K - Wi * Si; 0)$

where:

Si is the price of asset i in the basket.

Wi is the quantity of asset i in the basket.

K is the strike price.

Financial Derivatives Toolbox software supports Longstaff-Schwartz and Nengiu Ju models for pricing basket options. The Longstaff-Schwartz model supports both European, Bermuda, and American basket options. The Nengiu Ju model only supports European basket options. If you want to price either an American or Bermuda basket option, use the functions for the Longstaff-Schwartz model. To price a European basket option, use either the functions for the Longstaff-Schwartz model or the Nengiu Ju model.

Function	Purpose
basketbyls	Price basket options using the Longstaff-Schwartz model.
basketsensbyls	Calculate price and sensitivities for basket options using the Longstaff-Schwartz model.
basketbyju	Price European basket options using the Nengjiu Ju approximation model.
basketsensbyju	Calculate European basket options price and sensitivity using the Nengjiu Ju approximation model.
basketstockspec	Specify a basket stock structure.

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 pay the first strike price and receives 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.

The following functions support compound options.

Function	Purpose
`compoundbycrr`	Price compound options from a CRR binomial tree.
`compoundbyeqp`	Price compound options from an EQP binomial tree.
`compoundbyitt`	Price compound options using an implied trinomial tree (ITT).
`instcompound`	Construct a compound option.

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 software 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).

The following functions support lookback options.

Function	Purpose
`lookbackbycrr`	Price lookback options from a CRR binomial tree.
`lookbackbyeqp`	Price lookback options from an EQP binomial tree.
`lookbackbyitt`	Price lookback options using an implied trinomial tree (ITT).
`instlookback`	Construct a lookback option.

Digital Option

A *digital* option is an option whose payoff is characterized as having only two potential values: a fixed payout, when the option is in the money or a zero payout otherwise. This is the case irrespective of how far the asset price at maturity is above (call) or below (put) the strike.

Digital options are attractive to sellers because they guarantee a known maximum loss in the event that the option is exercised. This overcomes a fundamental problem with the vanilla options, where the potential loss is unlimited. Digital options are attractive to buyers because the option payoff is a known constant amount, and this amount can be adjusted to provide the exact quantity of protection required.

Financial Derivatives Toolbox supports four types of digital options:

- Cash-or-nothing option — Pays some fixed amount of cash if the option expires in the money.
- Asset-or-nothing option — Pays the value of the underlying security if the option expires in the money.

- Gap option — One strike decides if the option is in or out of money; another strike decides the size the size of the payoff.
- Supershare — Pays out a proportion of the assets underlying a portfolio if the asset lies between a lower and an upper bound at the expiry of the option.

The following functions calculate pricing and sensitivity for digital options.

Function	Purpose
`cashbybls`	Calculate the price of cash-or-nothing digital options using the Black-Scholes model.
`assetbybls`	Calculate the price of asset-or-nothing digital options using the Black-Scholes model.
`gapbybls`	Calculate the price of gap digital options using the Black-Scholes model.
`supersharebybls`	Calculate the price of supershare digital options using the Black-Scholes model.
`cashsensbybls`	Calculate the price and sensitivities of cash-or-nothing digital options using the Black-Scholes model.
`assetsensbybls`	Calculate the price and sensitivities of asset-or-nothing digital options using the Black-Scholes model.
`gapsensbybls`	Calculate the price and sensitivities of gap digital options using the Black-Scholes model.
`supersharesensbybls`	Calculate the price and sensitivities of supershare digital options using the Black-Scholes model.

Rainbow Option

A rainbow option payoff depends on the relative price performance of two or more assets. A *rainbow* option gives the holder the right to buy or sell the best or worst of two securities, or options that pay the best or worst of two assets.

Rainbow options are popular because of the lower premium cost of the structure relative to the purchase of two separate options. The lower cost reflects the fact that the payoff is generally lower than the payoff of the two separate options.

Financial Derivatives Toolbox supports two types of rainbow options:

- Minimum of two assets — The option holder has the right to buy(sell) one of two risky assets, whichever one is worth less.
- Maximum of two assets — The option holder has the right to buy(sell) one of two risky assets, whichever one is worth more.

The following rainbow options speculate/hedge on two equity assets.

Function	Purpose
`minassetbystulz`	Calculate the European rainbow option price on minimum of two risky assets using the Stulz option pricing model.
`minassetsensbystulz`	Calculate the European rainbow option prices and sensitivities on minimum of two risky assets using the Stulz pricing model.
`maxassetbystulz`	Calculate the European rainbow option price on maximum of two risky assets using the Stulz option pricing model.
`maxassetsensbystulz`	Calculate the European rainbow option prices and sensitivities on maximum of two risky assets using the Stulz pricing model.

Vanilla Option

A *vanilla option* is a category of options that includes only the most standard components. A vanilla option has an expiration date and straightforward strike price. American-style options and European-style options are both categorized as vanilla options.

The payoff for a vanilla option is as follows:

- For a call: $\max(St - K, 0)$
- For a put: $\max(K - St, 0)$

where:

St is the price of the underlying stock at time t.

K is the strike price.

The following functions support specifying or pricing a vanilla option.

Function	Purpose
`optstockbycrr`	Calculate the price of a European, Bermuda, or American stock option using a CRR tree.
`optstockbyeqp`	Calculate the price of a European, Bermuda, or American stock option using an EQP tree.
`optstockbyitt`	Calculate the price of a European, Bermuda, or American stock option using an ITT tree.
`instoptstock`	Specify a European or Bermuda option.

Bermuda Put and Call Schedule

A Bermuda option resembles a hybrid of American and European options. You exercise it on predetermined dates only, usually monthly. In Financial Derivatives Toolbox software, you indicate the relevant information for a Bermuda option in two input matrices:

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

Computing Prices and Sensitivities for Equity Derivatives Using Trees

In this section...

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
   HWTree            1x1              5904  struct
   ITTInstSet        1x1             12438  struct
   ITTTree           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     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 `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
   ITTInstSet        1x1             12438  struct
   ITTTree           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
```

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

```
instdisp(ITTInstSet)
```

```
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(ITTTree, ITTInstSet)

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 eight 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] = ittsens(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] = ittsens(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 many 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 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=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 ittsens 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 2-75 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.

Equity Derivatives Using Closed-Form Solutions

In this section...

Introduction

Financial Derivatives Toolbox software supports four types of closed-form solutions and analytical approximations to calculate price and sensitivities (greeks) of vanilla options:

- Black-Scholes model
- Black model
- Roll-Geske-Whaley model
- Bjerksund-Stensland 2002 model

Black-Scholes Model

The Black-Scholes model is one of the most commonly used models to price European calls and puts. It serves as a basis for many closed-form solutions used for pricing options. The standard Black-Scholes model is based on the following assumptions:

- There are no dividends paid during the life of the option.
- The option can only be exercised at maturity.
- The markets operate under a Markov process in continuous time.
- No commissions are paid.

- The risk-free interest rate is known and constant.
- Returns on the underlying stocks are log-normally distributed.

Note The Black-Scholes model implemented in Financial Derivatives Toolbox software allows dividends. The following three dividend methods are supported:

- Cash dividend
- Continuous dividend yield
- Constant dividend yield

However, not all Black-Scholes closed-form pricing functions support all three dividend methods. For more information on specifying the dividend methods, see `stockspec`.

Closed-form solutions based on a Black-Scholes model support the following tasks.

Task	Function
Price European options with different dividends using the Black-Scholes option pricing model.	`optstockbybls`
Calculate European option prices and sensitivities using the Black-Scholes option pricing model.	`optstocksensbybls`
Calculate implied volatility on European options using the Black-Scholes option pricing model.	`impvbybls`
Price European simple chooser options using Black-Scholes model.	`chooserbybls`

For an example using the Black-Scholes model, see “Computing Prices and Sensitivities Using the Black-Scholes Model” on page 3-54.

Black Model

Use the Black model for pricing European options on physical commodities, forwards or futures. The Black model supported by Financial Derivatives Toolbox software is a special case of the Black-Scholes model. The Black model uses a forward price as an underlier in place of a spot price. The assumption is that the forward price at maturity of the option is log-normally distributed.

Closed-form solutions for a Black model support the following tasks.

Task	Function
Price European options on futures using the Black option pricing model.	`optstockbyblk`
Calculate European option prices and sensitivities on futures using the Black option pricing model.	`optstocksensbyblk`
Calculate implied volatility for European options using the Black option pricing model.	`impvbyblk`

For an example using the Black model, see "Computing Prices and Sensitivities Using the Black Model" on page 3-56.

Roll-Geske-Whaley Model

Use the Roll-Geske-Whaley approximation method to price American call options paying a single cash dividend. This model is based on the modification of the observed stock price for the present value of the dividend and also supports a compound option to account for the possibility of early exercise. The Roll-Geske-Whaley model has drawbacks due to an escrowed dividend price approach which may lead to arbitrage. For further explanation, see *Options, Futures, and Other Derivatives* by John Hull.

Closed-form solutions for a Roll-Geske-Whaley model support the following tasks.

Task	Function
Price American call options with a single cash dividend using the Roll-Geske-Whaley option pricing model.	`optstockbyrgw`
Calculate American call prices and sensitivities using the Roll-Geske-Whaley option pricing model.	`optstocksensbyrgw`
Calculate implied volatility for American call options using the Roll-Geske-Whaley option pricing model.	`impvbyrgw`

For an example using the Roll-Geske-Whaley model, see "Computing Prices and Sensitivities Using the Roll-Geske-Whaley Model" on page 3-57.

Bjerksund-Stensland 2002 Model

Use the Bjerksund-Stensland 2002 model for pricing American puts and calls with continuous dividend yield. This model works by dividing the time to maturity of the option in two separate parts, each with its own flat exercise boundary (trigger price). The Bjerksund-Stensland 2002 method is a generalization of the Bjerksund and Stensland 1993 method and is considered to be computationally efficient . For further explanation, see *Closed Form Valuation of American Options* by Bjerksund and Stensland.

Closed-form solutions for a Bjerksund-Stensland 2002 model support the following tasks.

Task	Function
Price American options with continuous dividend yield using the Bjerksund-Stensland 2002 option pricing model.	`optstockbybjs`
Calculate American options prices and sensitivities using the Bjerksund-Stensland 2002 option pricing model.	`optstocksensbybjs`

Task	Function
Calculate implied volatility for American options using the Bjerksund-Stensland 2002 option pricing model.	`impvbybjs`

For an example using the Bjerksund-Stensland 2002 model, see "Computing Prices and Sensitivities Using the Bjerksund-Stensland Model" on page 3-58.

Computing Prices and Sensitivities Using the Black-Scholes Model

Consider a European stock option with an exercise price of $40 on January 1, 2008 that expires on July 1, 2008. Assume the underlying stock pays dividends of $0.50 on March 1 and June 1. The stock is trading at $40 and has a volatility of 30% per annum. The risk-free rate is 4% per annum. Using this data, calculate the price of a call and a put option on the stock using the Black-Scholes option pricing model:

```
Strike = 40;
AssetPrice = 40;
Sigma = .3;
Rates = 0.04;
Settle = 'Jan-01-08';
Maturity = 'Jul-01-08';

Div1 = 'March-01-2008';
Div2 = 'Jun-01-2008';
```

Create `RateSpec` and `StockSpec`:

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


StockSpec = stockspec(Sigma, AssetPrice, {'cash'}, 0.50,{Div1,Div2});
```

Define two options, one call and one put:

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

Calculate the price of the European options:

```
Price = optstockbybls(RateSpec, StockSpec, Settle, Maturity, OptSpec, Strike)

Price =

    3.2063
    3.4027
```

The first element of the `Price` vector represents the price of the call ($3.21); the second is the price of the put ($3.40). Use the function `optstocksensbybls` to compute six sensitivities for the Black-Scholes model: `delta`, `gamma`, `vega`, `lambda`, `rho`, and `theta` and the `price` of the option.

The selection of output parameters and their order is determined by the optional input parameter `OutSpec`. This parameter is a cell array of strings, each one specifying a desired output parameter. The order in which these output parameters are returned by the function is the same as the order of the strings contained in `OutSpec`.

As an example, consider the same options as the previous example. To calculate their `Delta`, `Rho`, `Price`, and `Gamma`, build the cell array `OutSpec` as follows:

```
OutSpec = {'delta', 'rho', 'price', 'gamma'};

[Delta, Rho, Price, Gamma] =optstocksensbybls(RateSpec, StockSpec, Settle,...
Maturity, OptSpec, Strike, 'OutSpec', OutSpec)

Delta =

    0.5328
   -0.4672


Rho =

    8.7902
  -10.8138


Price =
```

```
    3.2063
    3.4027


Gamma =

    0.0480
    0.0480
```

Computing Prices and Sensitivities Using the Black Model

Consider two European call options on a futures contract with exercise prices of $20 and $25 that expire on September 1, 2008. Assume that on May 1, 2008 the contract is trading at $20 and has a volatility of 35% per annum. The risk-free rate is 4% per annum. Using this data, calculate the price of the call futures options using the Black model:

```
Strike = [20; 25];
AssetPrice = 20;
Sigma = .35;
Rates = 0.04;
Settle = 'May-01-08';
Maturity = 'Sep-01-08';
```

Create `RateSpec` and `StockSpec`:

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

StockSpec = stockspec(Sigma, AssetPrice);
```

Define the call option:

```
OptSpec = {'call'};
```

Calculate price and all sensitivities of the European futures options:

```
OutSpec = {'All'}
```

```
[Delta, Gamma, Vega, Lambda, Rho, Theta, Price] = optstocksensbyblk(RateSpec,...
StockSpec, Settle, Maturity, OptSpec, Strike, 'OutSpec', OutSpec);

Price =

    1.5903
    0.3037
```

The first element of the `Price` vector represents the price of the call with an exercise price of $20 ($1.59); the second is the price of the call with an exercise price of $25 ($2.89).

The function `impvbyblk` is used to compute the implied volatility using the Black option pricing model. Assuming that the previous European call futures are trading at $1.5903 and $0.3037, you can calculate their implied volatility:

```
Volatility = impvbyblk(RateSpec, StockSpec, Settle, Maturity,...
OptSpec,  Strike, Price);
```

As expected, you get volatilities of 35%. If the call futures were trading at $1.50 and $0.50 in the market, the implied volatility would be 33% and 42%:

```
Volatility = impvbyblk(RateSpec, StockSpec, Settle, Maturity,...
OptSpec,  Strike, [1.50;0.5])

Volatility =

    0.3301
    0.4148
```

Computing Prices and Sensitivities Using the Roll-Geske-Whaley Model

Consider two American call options, with exercise prices of $110 and $100 on June 1, 2008, that expire on June 1, 2009. Assume the underlying stock pays dividends of $0.001 on December 1, 2008. The stock is trading at $80 and has a volatility of 20% per annum. The risk-free rate is 6% per annum. Using this data, calculate the price of the American calls using the Roll-Geske-Whaley option pricing model:

```
AssetPrice = 80;
```

```
Settle = 'Jun-01-2008';
Maturity = 'Jun-01-2009';
Strike = [110; 100];

Rate = 0.06;
Sigma  = 0.2;

DivAmount = 0.001;
DivDate = 'Dec-01-2008';
```

Create `RateSpec` and `StockSpec`:

```
StockSpec = stockspec(Sigma, AssetPrice, {'cash'}, DivAmount, DivDate);

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

Calculate the call prices:

```
Price  = optstockbyrgw(RateSpec, StockSpec, Settle, Maturity, Strike)

Price =

    0.8398
    2.0236
```

The first element of the `Price` vector represents the price of the call with an exercise price of $110 ($0.84); the second is the price of the call with an exercise price of $100 ($2.02).

Computing Prices and Sensitivities Using the Bjerksund-Stensland Model

Consider four American stock options (two calls and two puts) with an exercise price of $100 that expire on July 1, 2008. Assume the underlying stock pays a continuous dividend yield of 4% as of January 1, 2008. The stock has a volatility of 20% per annum and the risk-free rate is 8% per annum. Using this data, calculate the price of the American calls and puts assuming the following current prices of the stock: $80, $90 (for the calls) and $100 and $110 (for the puts):

```
Settle = 'Jan-1-2008';
Maturity = 'Jul-1-2008';
Strike = 100;
AssetPrice = [80; 90; 100; 110];
DivYield = 0.04;

Rate = 0.08;
Sigma = 0.20;
```

Create `RateSpec` and `StockSpec`:

```
StockSpec = stockspec(Sigma, AssetPrice, {'continuous'}, DivYield);

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

Define the option type:

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

Compute the option prices:

```
Price = optstockbybjs(RateSpec, StockSpec, Settle, Maturity, OptSpec, Strike)

Price =

    0.4144
    2.1804
    4.7253
    1.7164
```

The first two elements of the `Price` vector represent the price of the calls ($0.41 and $2.18), the last two elements represent the price of the put options ($4.72 and $1.72). Use the function `optstocksensbybjs` to compute six sensitivities for the Bjerksund-Stensland model: `delta`, `gamma`, `vega`, `lambda`, `rho`, and `theta` and the `price` of the option. The selection of output parameters and their order is determined by the optional input parameter `OutSpec`. This parameter is a cell array of strings, each one specifying a desired output parameter. The order in which these output parameters are returned by the function is the same as the order of the strings contained in `OutSpec`. As an example, consider the same options as the previous example.

To calculate their `delta`, `gamma`, and `price`, build the cell array `OutSpec` as follows:

```
OutSpec = {'delta', 'gamma', 'price'};
```

The outputs of `optstocksensbybjs` will be in the same order as in `OutSpec`.

```
[Delta ,Gamma, Price]= optstocksensbybjs(RateSpec, StockSpec, Settle,...
Maturity, OptSpec, Strike, 'OutSpec', OutSpec)

Delta =

    0.0843
    0.2912
    0.4803
    0.2261

Gamma =

    0.0136
    0.0267
    0.0304
    0.0217

Price =

    0.4144
    2.1804
    4.7253
    1.7164
```

4

Hedging 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 required 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

In this section...

Introduction

Financial Derivatives Toolbox software 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 4-12.

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`, `eqpsens`, or `ittsens` 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 2-33. Likewise, for examples showing the computation of sensitivities for equity exotic options, see "Computing Instrument Sensitivities" on page 3-44.

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.

```
Value0 = Holdings' * Price

Value0 =

     23674.62

Value1 = Quantity * Price

Value1 =

     23674.62
```

Partially Hedged Portfolio

Building on the example in "Maintaining Existing Allocations" on page 4-6, 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 4-7 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 0s and call `hedgeopt` again. The following example uses data from "Hedging with hedgeopt" on page 4-4.

```
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 required 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 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.

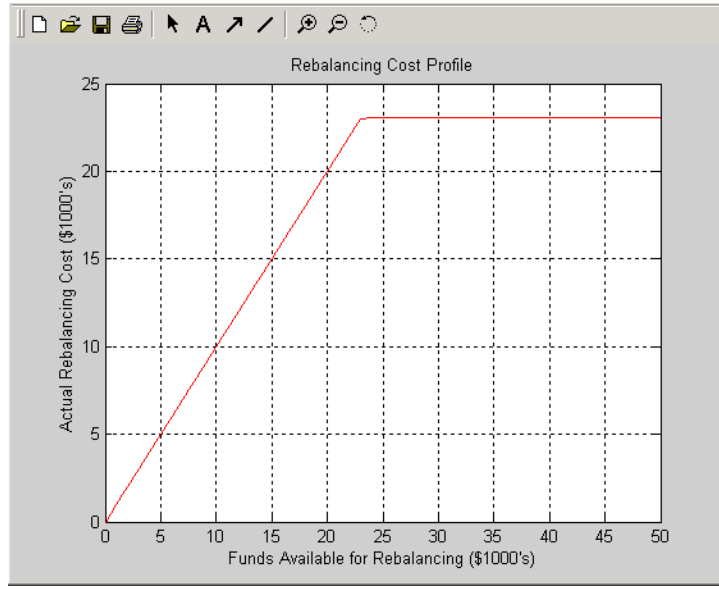
```
MaxCost = [0:1000:50000];
```

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)

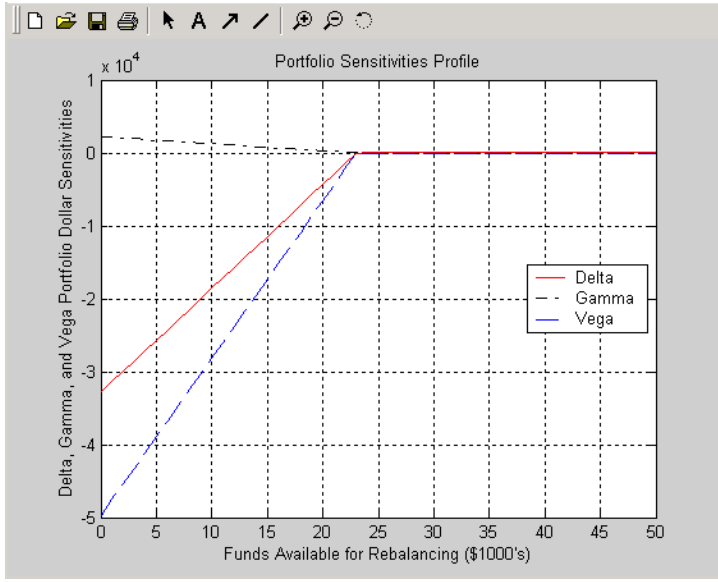
```
plot(MaxCost/1000, Cost/1000, 'red'), grid
xlabel('Funds Available for Rebalancing ($1000''s)')
ylabel('Actual Rebalancing Cost ($1000''s)')
title ('Rebalancing Cost Profile')
```



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 4-11 and Funds Available for Rebalancing on page 4-12 indicate that there is no benefit 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.

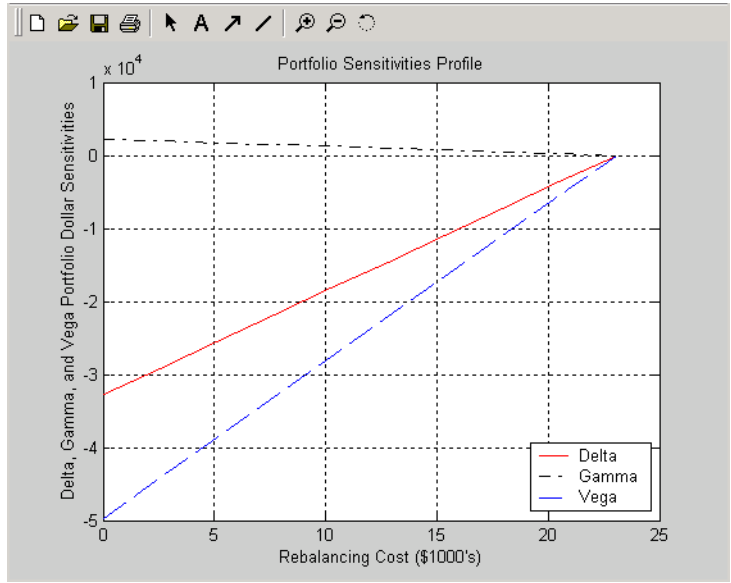
There is, in fact, a third calling syntax available for `hedgeopt` directly related to the results shown above for `hedgeslf`. Suppose, instead of directly specifying the funds available for rebalancing (the most money you are willing to spend), you want to simply specify the number of points along the 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 `hedgeslf` internally to determine the maximum cost needed to minimize the portfolio sensitivities ($23,055.90), and evenly samples the cost frontier between $0 and $23,055.90.

Note that both `hedgeopt` and `hedgeslf` cast the optimization problem as a constrained linear least squares problem. Depending 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 4-21 for additional information.

Specifying Constraints with ConSet

In this section...

Introduction

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 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 required 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
```

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

```
Value0 - Value1

ans =

      22819.52
```

Portfolio Rebalancing

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

For reference, startup `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 4-15.

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

In this section...

Overview

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

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

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

Depending on the constraint 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 needs 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 $Z \le 1$. Furthermore, assume you somehow add another constraint that effectively restricts $Z \le 0$. The constraint $Z \le 1$ 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 4-11, Funds Available for Rebalancing on page 4-12, and Rebalancing Cost on page 4-15.

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
```

`hedgeslf` 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 `hedgeslf` with `ConSet` as the last input.

```
[Sens, Cost, Quantity1] = hedgeslf(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 `hedgeslf` enforces the upper bound on the fourth instrument, and the portfolio continues to be fully hedged and self-financing.

5

Function Reference

Implied Trinomial Tree Utilities (p. 5-10)	Work with implied trinomial tree utilities
Hull-White Utilities (p. 5-11)	Work with Hull-White utilities
Tree Manipulation (p. 5-11)	General tree manipulation
Derivatives Pricing Options (p. 5-12)	Work with derivatives pricing options
Pricing and Sensitivity Using Black-Scholes Option Pricing Model (p. 5-12)	Work with Black-Scholes Option Pricing
Pricing and Sensitivity Using Black Option Pricing Model (p. 5-13)	Work with Black Option Pricing
Pricing and Sensitivity Using Longstaff-Schwartz Option Pricing Model (p. 5-14)	Work with Longstaff-Schwartz Option Pricing
Pricing and Sensitivity Using Nengjiu Ju Approximation Model (p. 5-14)	Work with Nengjiu Ju Option Pricing
Pricing and Sensitivity Using Role-Geske-Whaley Option Pricing Model (p. 5-15)	Work with Role-Geske-Whaley Option Pricing
Pricing and Sensitivity Using Bjerksund-Stensland Option Pricing Model (p. 5-15)	Work with Bjerksund-Stensland Option Pricing
Pricing and Sensitivity Using Stulz Option Pricing Model (p. 5-16)	Work with Bjerksund-Stensland Option Pricing
Instrument Portfolio Handling (p. 5-16)	Work with instrument portfolios
Financial Object Structures (p. 5-18)	Work with financial structures
Interest Term Structure (p. 5-18)	Work with interest term structure
Date (p. 5-18)	Display date entries
Graphical Display (p. 5-19)	Display tree information graphically

Portfolio Hedge Allocation

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

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

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
`swaptionbyhjm`	Price swaption from HJM interest-rate tree

Black-Derman-Toy Trees

`bdtprice`	Instrument prices from BDT interest-rate tree
`bdtsens`	Instrument prices and sensitivities from BDT interest-rate tree
`bdttimespec`	Specify time structure for BDT interest-rate tree
`bdttree`	Construct BDT interest-rate tree
`bdtvolspec`	Specify BDT interest-rate volatility process

Black-Karasinski Trees

`bkprice`	Instrument prices from Black-Karasinski interest-rate tree
`bksens`	Instrument prices and sensitivities from Black-Karasinski interest-rate tree

`bktimespec`	Specify time structure for Black-Karasinski tree
`bktree`	Construct Black-Karasinski interest-rate tree
`bkvolspec`	Specify Black-Karasinski interest-rate volatility process
`swaptionbybk`	Price swaption from BK interest-rate tree

Cox-Ross-Rubinstein Trees

`crrprice`	Instrument prices from CRR tree
`crrsens`	Instrument prices and sensitivities from CRR tree
`crrtimespec`	Specify time structure for CRR tree
`crrtree`	Construct CRR stock tree

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

Hull-White Trees

`hwcalbycap`	Calibrate Hull-White tree using caps
`hwcalbyfloor`	Calibrate Hull-White tree using floors
`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

Implied Trinomial Tree

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

Heath-Jarrow-Morton Utilities

`bondbyhjm`	Price bond from HJM interest-rate tree
`capbyhjm`	Price cap instrument from HJM interest-rate tree
`cfbyhjm`	Price cash flows from HJM interest-rate tree
`fixedbyhjm`	Price fixed-rate note from HJM interest-rate tree
`floatbyhjm`	Price floating-rate note from HJM interest-rate tree
`floorbyhjm`	Price floor instrument from HJM interest-rate tree
`mmktbyhjm`	Create money-market tree from HJM interest-rate tree
`optbndbyhjm`	Price bond option from HJM interest-rate tree
`optembndbyhjm`	Price bonds with embedded options by Heath-Jarrow-Morton 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
`optembndbybdt`	Price bonds with embedded options by Black-Derman-Toy interest rate tree
`swapbybdt`	Price swap instrument from BDT interest-rate tree
`swaptionbybdt`	Price swaption 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
`optembndbybk`	Price bonds with embedded options by 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 implied trinomial tree (ITT)
`barrierbyitt`	Price barrier options using implied trinomial tree (ITT)
`compoundbyitt`	Price compound options using implied trinomial tree (ITT)
`lookbackbyitt`	Price lookback option using implied trinomial tree (ITT)
`optstockbyitt`	Price options on stocks using 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
`optembndbyhw`	Price bonds with embedded options by Hull-White interest-rate tree
`swapbyhw`	Price swap instrument from Hull-White interest-rate tree
`swaptionbyhw`	Price swaption from HW 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

Pricing and Sensitivity Using Black-Scholes Option Pricing Model

`assetbybls`	Calculate price of asset-or-nothing digital options using Black-Scholes model
`assetsensbybls`	Calculate price and sensitivities of asset-or-nothing digital options using Black-Scholes model
`cashbybls`	Calculate price of cash-or-nothing digital options using Black-Scholes model

`cashsensbybls`	Calculate price and sensitivities of cash-or-nothing digital options using Black-Scholes model
`chooserbybls`	Price European simple chooser options using Black-Scholes model
`gapbybls`	Calculate price of gap digital options using Black-Scholes model
`gapsensbybls`	Calculate price and sensitivities of gap digital options using Black-Scholes model
`impvbybls`	Calculate implied volatility using Black-Scholes option pricing model
`optstockbybls`	Price options using Black-Scholes option pricing model
`optstocksensbybls`	Calculate option prices and sensitivities using Black-Scholes option pricing model
`supersharebybls`	Calculate price of supershare digital options using Black-Scholes model
`supersharesensbybls`	Calculate price and sensitivities of supershare digital options using Black-Scholes model

Pricing and Sensitivity Using Black Option Pricing Model

`capbyblk`	Price caps using Black option pricing model
`floorbyblk`	Price floors using Black option pricing model
`impvbyblk`	Calculate implied volatility using Black option pricing model

`optstockbyblk`	Price options on futures using Black option pricing model
`optstocksensbyblk`	Calculate option prices and sensitivities on futures using Black pricing model

Pricing and Sensitivity Using Longstaff-Schwartz Option Pricing Model

`basketbyls`	Price basket options using Longstaff-Schwartz model
`basketsensbyls`	Calculate price and sensitivities for basket options using Longstaff-Schwartz model
`basketstockspec`	Specify basket stock structure

Pricing and Sensitivity Using Nengjiu Ju Approximation Model

`basketbyju`	Price European basket options using Nengjiu Ju approximation model
`basketsensbyju`	Calculate European basket options price and sensitivities using Nengjiu Ju approximation model

Pricing and Sensitivity Using Role-Geske-Whaley Option Pricing Model

`impvbyrgw`	Calculate implied volatility using Roll-Geske-Whaley option pricing model for American call option
`optstockbyrgw`	Calculate American call option prices using Roll-Geske-Whaley option pricing model
`optstocksensbyrgw`	Calculate American call option prices and sensitivities using Roll-Geske-Whaley option pricing model

Pricing and Sensitivity Using Bjerksund-Stensland Option Pricing Model

`impvbybjs`	Calculate implied volatility using Bjerksund-Stensland 2002 option pricing model
`optstockbybjs`	Price American options using Bjerksund-Stensland 2002 option pricing model
`optstocksensbybjs`	Calculate American option prices and sensitivities using Bjerksund-Stensland 2002 option pricing model

Pricing and Sensitivity Using Stulz Option Pricing Model

`maxassetbystulz`	Calculate European rainbow option price on maximum of two risky assets using Stulz option pricing model
`maxassetsensbystulz`	Calculate European rainbow option prices and sensitivities on maximum of two risky assets using Stulz pricing model
`minassetbystulz`	Calculate European rainbow option prices on minimum of two risky assets using Stulz option pricing model
`minassetsensbystulz`	Calculate European rainbow option prices and sensitivities on minimum of two risky assets using Stulz pricing model

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
`instfields`	List field names
`instfind`	Search instruments for matching conditions
`instfixed`	Construct fixed-rate instrument
`instfloat`	Construct floating-rate instrument
`instfloor`	Construct floor instrument
`instget`	Data from instrument variable
`instgetcell`	Data and context from instrument variable
`instlength`	Count instruments
`instlookback`	Construct lookback option
`instoptbnd`	Construct bond option
`instoptembnd`	Constructor for `'Type'`, `'OptEmBond'` bond with embedded option
`instoptstock`	Construct stock option
`instselect`	Create instrument subset by matching conditions
`instsetfield`	Add or reset data for existing instruments
`instswap`	Construct swap instrument
`instswaption`	Construct swaption instrument
`insttypes`	List types

Financial Object Structures

`classfin`	Create financial structure or return financial structure class name
`isafin`	True if input argument is financial structure type or financial object class
`stockspec`	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

6

Functions — Alphabetical List

Purpose

Price Asian option from CRR binomial tree

Syntax

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

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 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. 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 implied trinomial tree (ITT)

Syntax

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

Arguments

`ITTTree`	Stock tree structure created by `itttree`.
`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(ITTTree, 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 `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.

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

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

```
Price = asianbyitt(ITTTree, 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

Calculate price of asset-or-nothing digital options using Black-Scholes model

Syntax

```
Price = assetbybls(RateSpec, StockSpec, Settle, Maturity,
OptSpec, Strike)
```

Arguments

`RateSpec`	The annualized continuously compounded rate term structure. For information on the interest rate specification, see `intenvset`.
`StockSpec`	Stock specification. See `stockspec`.
`Settle`	`NINST`-by-`1` vector of settlement or trade dates.
`Maturity`	`NINST`-by-`1` vector of maturity dates.
`OptSpec`	`NINST`-by-`1` cell array of strings `'call'` or `'put'`.
`Strike`	`NINST`-by-`1` vector of payoff strike price values.

Description

`Price = assetbybls(RateSpec, StockSpec, Settle, Maturity, OptSpec, Strike)` computes asset-or-nothing option prices using the Black-Scholes option pricing model.

`Price` is a `NINST`-by-`1` vector of expected option prices.

Examples

Consider two asset-or-nothing put options on a nondividend paying stock with a strike of 95 and 93 and expiring on January 30, 2009. On November 3, 2008 the stock is trading at 97.50. Using this data, calculate the price of the asset-or-nothing put options if the risk-free rate is 4.5% and the volatility is 22%.

Create the `RateSpec`:

```
Settle = 'Nov-3-2008';
Maturity = 'Jan-30-2009';
Rates = 0.045;
```

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

Define the StockSpec:

```
AssetPrice = 97.50;
Sigma = .22;
StockSpec = stockspec(Sigma, AssetPrice);
```

Define the put options:

```
OptSpec = {'put'};
Strike = [95;93];
```

Calculate the price:

```
Paon = assetbybls(RateSpec, StockSpec, Settle, Maturity, OptSpec, Strike)

Paon =

   33.7666
   26.9662
```

See Also

assetsensbybls, cashbybls, gapbybls, supersharebybls

Purpose

Calculate price and sensitivities of asset-or-nothing digital options using Black-Scholes model

Syntax

```
PriceSens = assetsensbybls(RateSpec, StockSpec, Settle,
Maturity, OptSpec, Strike)
PriceSens = assetsensbybls(RateSpec, StockSpec, Settle,
Maturity, OptSpec, Strike, OutSpec)
```

Arguments

`RateSpec`	The annualized, continuously compounded rate term structure. For information on the interest rate specification, see `intenvset`.
`StockSpec`	Stock specification. See `stockspec`.
`Settle`	`NINST`-by-1 vector of settlement or trade dates.
`Maturity`	`NINST`-by-1 vector of maturity dates.
`OptSpec`	`NINST`-by-1 cell array of strings `'call'` or `'put'`.
`Strike`	`NINST`-by-1 vector of strike price values.
`OutSpec`	(Optional) All optional inputs are specified as matching parameter name/value pairs. The parameter name is specified as a character string, followed by the corresponding parameter value. You can specify parameter name/value pairs in any order. Names are case-insensitive and partial string matches are allowed provided no ambiguities exist. Valid parameter names are: • `NOUT`-by-1 or 1-by-`NOUT` cell array of strings indicating the nature and order of the outputs for the function. Possible values are `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, or `'All'`.

For example, `OutSpec = {'Price'; 'Lamba'; 'Rho'}` specifies that the output should be `Price`, `Lambda`, and `Rho`, in that order.

To invoke from a function: `[Price, Lambda, Rho] = assetsensbybls(..., 'OutSpec', {'Price', 'Lamba', 'Rho'})`

`OutSpec = {'All'}` specifies that the output should be `Delta`, `Gamma`, `Vega`, `Lambda`, `Rho`, `Theta`, and `Price`, in that order. This is the same as specifying `OutSpec` as `OutSpec = {'Delta', 'Gamma', 'Vega', 'Lambda', 'Rho', 'Theta', 'Price'};`.

- Default is `OutSpec = {'Price'}`.

Description

`PriceSens = assetsensbybls(RateSpec, StockSpec, Settle, Maturity, OptSpec, Strike)` computes asset-or-nothing option prices using the Black-Scholes option pricing model.

`PriceSens = assetsensbybls(RateSpec, StockSpec, Settle, Maturity, OptSpec, Strike, OutSpec)` includes the parameter/value pairs defined for `OutSpec`, and computes asset-or-nothing option prices and sensitivities using the Black-Scholes option pricing model.

`PriceSens` is a `NINST`-by-`1` vector of expected option prices and sensitivities.

Examples

Consider two asset-or-nothing put options on a nondividend paying stock with a strike of 95 and 93 and expiring on January 30, 2009. On November 3, 2008 the stock is trading at 97.50. Using this data, calculate the price and sensitivity of the asset-or-nothing put options if the risk-free rate is 4.5% and the volatility is 22%.

Create the `RateSpec`:

```
Settle = 'Nov-3-2008';
```

```
Maturity = 'Jan-30-2009';
Rates = 0.045;
Compounding = -1;
RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle,...
'EndDates', Maturity, 'Rates', Rates, 'Compounding', Compounding);
```

Define the StockSpec:

```
AssetPrice = 97.50;
Sigma = .22;
StockSpec = stockspec(Sigma, AssetPrice);
```

Define the put options:

```
OptSpec = {'put'};
Strike = [95;93];
```

Calculate the delta, price, and gamma:

```
OutSpec = { 'delta';'price';'gamma'};
[Delta, Price, Gamma] = assetsensbybls(RateSpec, StockSpec, Settle,...
Maturity, OptSpec, Strike, 'OutSpec', OutSpec)

Delta =

   -3.0833
   -2.8337


Price =

   33.7666
   26.9662


Gamma =

    0.0941
```

```
0.1439
```

See Also

assetbybls

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.

`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.

```
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 implied trinomial tree (ITT)

Syntax

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

Arguments

`ITTTree`	Stock tree structure created by `itttree`.
`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.

`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(ITTTree, 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 `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.

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

Price = barrierbyitt(ITTTree,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`

basketbyju

Purpose

Price European basket options using Nengjiu Ju approximation model

Syntax

```
Price = basketbyju(RateSpec, BasketStockSpec, OptSpec, Strike,
Settle, Maturity)
```

Description

`Price = basketbyju(RateSpec, BasketStockSpec, OptSpec, Strike, Settle, Maturity)` prices European basket options using the Nengjiu Ju approximation model.

Input Arguments

`RateSpec`

Annualized, continuously compounded rate term structure. For more information on the interest rate specification, see `intenvset`.

`BasketStockSpec`

`BasketStock` specification. For information on the basket of stocks specification, see `basketstockspec`.

`OptSpec`

String or 2-by-1 cell array of the strings `'call'` or `'put'`.

`Strike`

Scalar for the option strike price.

`Settle`

Scalar of the settlement or trade date specified as a string or serial date number.

`Maturity`

Maturity date specified as a string or serial date number.

Output Arguments

`Price`

Price of the basket option.

Examples

Find a European call basket option of two stocks. Assume that the stocks are currently trading at $10 and $11.50 with annual volatilities

of 20% and 25%, respectively. The basket contains one unit of the first stock and one unit of the second stock. The correlation between the assets is 30%. On January 1, 2009, an investor wants to buy a 1-year call option with a strike price of $21.50. The current annualized, continuously compounded interest rate is 5%. Use this data to compute the price of the call basket option with the Ju approximation model.

```
Settle = 'Jan-1-2009';
Maturity  = 'Jan-1-2010';

% Define RateSpec
Rate = 0.05;
Compounding = -1;
RateSpec = intenvset('ValuationDate', Settle, 'StartDates', ...
Settle, 'EndDates', Maturity, 'Rates', Rate, 'Compounding', Compounding);

% Define the Correlation matrix. Correlation matrices are symmetric, and
% have ones along the main diagonal.
Corr = [1 0.30; 0.30 1];

% Define BasketStockSpec
AssetPrice =  [10;11.50];
Volatility = [0.2;0.25];
Quantity = [1;1];
BasketStockSpec = basketstockspec(Volatility, AssetPrice, Quantity, Corr);

%Compute the price of the call basket option
OptSpec = {'call'};
Strike = 21.5;
PriceCorr30 = basketbyju(RateSpec, BasketStockSpec, OptSpec, Strike, Settle, Maturity)
```

This returns:

```
PriceCorr30 =

   2.12214
```

Compute the price of the basket instrument for these two stocks with a correlation of 60%. Then compare this cost to the total cost of buying two individual call options:

```
Corr = [1 0.60; 0.60 1];

% Define the new BasketStockSpec
BasketStockSpec = basketstockspec(Volatility, AssetPrice, Quantity, Corr);

%Compute the price of the call basket option with Correlation = -0.60
PriceCorr60 = basketbyju(RateSpec, BasketStockSpec, OptSpec, Strike, Settle, Maturity)
```

This returns:

```
PriceCorr60 =

   2.27566
```

The following table summarizes the sensitivity of the option to correlation changes. In general, the premium of the basket option decreases with lower correlation and increases with higher correlation.

Correlation	-0.60	-0.30	0	0.30	0.60
Premium	1.52830	1.76006	1.9527	2.1221	2.2756

Compute the cost of two vanilla 1-year call options using the Black-Scholes (BLS) model on the individual assets:

```
StockSpec = stockspec(Volatility, AssetPrice);
StrikeVanilla= [10;11.5];

PriceVanillaOption = optstockbybls(RateSpec, StockSpec, Settle, Maturity,...
OptSpec, StrikeVanilla)
```

This returns:

```
PriceVanillaOption =
```

```
1.0451
1.4186
```

Find the total cost of buying two individual call options:

```
sum(PriceVanillaOption)
```

This returns:

```
ans=2.4637
```

The total cost of purchasing two individual call options is $2.4637, compared to the maximum cost of the basket option of $2.27 with a correlation of 60%.

References

Nengjiu Ju, "Pricing Asian and Basket Options Via Taylor Expansion", *Journal of Computational Finance*, Vol. 5, 2002.

See Also

`basketstockspec` | `basketsensbyju`

How To

- "Basket Option" on page 3-25

basketbyls

Purpose

Price basket options using Longstaff-Schwartz model

Syntax

```
Price = basketbyls(RateSpec, BasketStockSpec, OptSpec, Strike,
Settle, ExerciseDates)
Price = basketbyls(RateSpec, BasketStockSpec, OptSpec, Strike,
Settle, ExerciseDates, 'ParameterName', ParameterValue ...)
```

Description

`Price = basketbyls(RateSpec, BasketStockSpec, OptSpec, Strike, Settle, ExerciseDates)` prices basket options using the Longstaff-Schwartz model.

`Price = basketbyls(RateSpec, BasketStockSpec, OptSpec, Strike, Settle, ExerciseDates, 'ParameterName', ParameterValue ...)` accepts optional inputs as one or more comma-separated parameter/value pairs. `'ParameterName'` is the name of the parameter inside single quotes. `'ParameterValue` is the value corresponding to `'ParameterName'`. Specify parameter-value pairs in any order. Names are case-insensitive and partial string matches are allowable, if no ambiguities exist.

Input Arguments

`RateSpec`

Annualized, continuously compounded rate term structure. For more information on the interest rate specification, see `intenvset`.

`BasketStockSpec`

`BasketStock` specification. For information on the basket of stocks specification, see `basketstockspec`.

`OptSpec`

String or 2-by-1 cell array of the strings `'call'` or `'put'`.

`Strike`

The option strike price:

- For a European or Bermuda option, `Strike` is a scalar (European) or 1-by-`NSTRIKES` (Bermuda) vector of strike price.

- For an American option, `Strike` is a scalar vector of the strike price.

`Settle`

Scalar of the settlement or trade date specified as a string or serial date number.

`ExerciseDates`

The exercise date for the option:

- For a European or Bermuda option, `ExerciseDates` is a `1`-by-`1` (European) or `1`-by-`NSTRIKES` (Bermuda) vector of exercise dates. For a European option, there is only one `ExerciseDate` on the option expiry date.
- For an American option, `ExerciseDates` is a `1`-by-`2` vector of exercise date boundaries. The option exercises on any date between, or including, the pair of dates on that row. If there is only one non-`NaN` date, or if `ExerciseDates` is `1`-by-`1`, the option exercises between the `Settle` date and the single listed `ExerciseDate`.

Parameter–Value Pairs

`AmericanOpt`

Parameter values are a scalar flag.

- 0 — European/Bermuda
- 1 — American

Default: 0

`NumPeriods`

Parameter value is a scalar number of simulation periods per trial. `NumPeriods` is considered only when pricing European basket options. For American and Bermuda basket options, `NumPeriod` equals the number of exercise days during the life of the option.

Default: 100

`NumTrials`

Parameter value is a scalar number of independent sample paths (simulation trials).

Default: 1000

Output Arguments

`Price`

Price of the basket option.

Examples

Find an American call basket option of three stocks. The stocks are currently trading at $35, $40 and $45 with annual volatilities of 12%, 15% and 18%, respectively. The basket contains 33.33% of each stock. Assume the correlation between all pair of assets is 50%. On May 1, 2009, an investor wants to buy a three-year call option with a strike price of $42. The current annualized continuously compounded interest rate is 5%. Use this data to compute the price of the call basket option using the Longstaff-Schwartz model.

```
Settle = 'May-1-2009';
Maturity  = 'May-1-2012';

% Define RateSpec
Rate = 0.05;
Compounding = -1;
RateSpec = intenvset('ValuationDate', Settle, 'StartDates',...
Settle, 'EndDates', Maturity, 'Rates', Rate, 'Compounding', Compounding);

% Define the Correlation matrix. Correlation matrices are symmetric,
% and have ones along the main diagonal.
```

```
Corr = [1 0.50 0.50; 0.50 1 0.50;0.50 0.50 1];

% Define BasketStockSpec
AssetPrice =  [35;40;45];
Volatility = [0.12;0.15;0.18];
Quantity = [0.333;0.333;0.333];
BasketStockSpec = basketstockspec(Volatility, AssetPrice, Quantity, Corr);

% Compute the price of the call basket option
OptSpec = {'call'};
Strike = 42;
AmericanOpt = 1; % American option
Price = basketbyls(RateSpec, BasketStockSpec, OptSpec, Strike, Settle, Maturity,...
'AmericanOpt',AmericanOpt)
```

This returns:

```
Price =

    5.60499
```

Increase the number of simulation trials to 2000 to give the following results:

```
NumTrial = 2000;
Price = basketbyls(RateSpec, BasketStockSpec, OptSpec, Strike, Settle, Maturity,...
'AmericanOpt',AmericanOpt,'NumTrials',NumTrial)
Price =

   5.6665
```

References

Longstaff, F.A., and E.S. Schwartz, “Valuing American Options by Simulation: A Simple Least-Squares Approach”, *The Review of Financial Studies*, Vol. 14, No. 1, Spring 2001, pp. 113–147.

See Also

basketstockspec | basketsensbyls

How To

- “Basket Option” on page 3-25

Purpose

Calculate European basket options price and sensitivities using Nengjiu Ju approximation model

Syntax

```
PriceSens = basketsensbyju(RateSpec, BasketStockSpec, OptSpec,
   Strike,
Settle, Maturity)
PriceSens = basketsensbyju(RateSpec, BasketStockSpec, OptSpec,
   Strike,
Settle, Maturity, 'ParameterName', ParameterValue ...)
```

Description

`PriceSens = basketsensbyju(RateSpec, BasketStockSpec, OptSpec, Strike, Settle, Maturity)` calculates prices and sensitivities for basket options using the Nengjiu Ju approximation model.

`PriceSens = basketsensbyju(RateSpec, BasketStockSpec, OptSpec, Strike, Settle, Maturity, 'ParameterName', ParameterValue ...)` accepts optional inputs as one or more comma-separated parameter/value pairs. *'ParameterName'* is the name of the parameter inside single quotes. *'ParameterValue* is the value corresponding to *'ParameterName'*. Specify parameter-value pairs in any order. Names are case-insensitive and partial string matches are allowable, if no ambiguities exist.

Input Arguments

`RateSpec`

Annualized, continuously compounded rate term structure. For more information on the interest rate specification, see `intenvset`.

`BasketStockSpec`

`BasketStock` specification. For information on the basket of stocks specification, see `basketstockspec`.

`OptSpec`

String or 2-by-1 cell array of the strings `'call'` or `'put'`.

`Strike`

Scalar of the option strike price.

`Settle`

Scalar of the settlement or trade date specified as a string or serial date number.

`Maturity`

Maturity date, specified as a string or serial date number.

Parameter–Value Pairs

`OutSpec`

Parameter value is an `NOUT`-by-`1` or `1`-by-`NOUT` cell array of strings indicating the nature and order of the outputs for the function. Possible values are `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`. For example, `OutSpec = {'Price', 'Lamba', 'Rho'}` specifies that the output is `Price`, `Lambda`, and `Rho`, in that order.

`OutSpec = {'All'}` specifies that the output should be `Delta`, `Gamma`, `Vega`, `Lambda`, `Rho`, `Theta`, and `Price`, in that order. This is the same as specifying `OutSpec` as `OutSpec = {'Delta', 'Gamma', 'Vega', 'Lambda', 'Rho', 'Theta', 'Price'};`.

Default: `OutSpec = {'Price'}`

`UndIdx`

Scalar of the indice of the underlying instrument to compute the sensitivity.

Default: `UndIdx = []`

Output Arguments

`PriceSens`

Expected prices or sensitivities values for the basket option.

Examples

Find a European call basket option of five stocks. Assume that the basket contains:

- 5% of the first stock trading at $110
- 15% of the second stock trading at $75
- 20% of the third stock trading at $40
- 25% of the fourth stock trading at $125
- 35% of the fifth stock trading at $92

These stocks have annual volatilities of 20% and the correlation between the assets is zero. On May 1, 2009, an investor wants to buy a 1-year call option with a strike price of $90. The current annualized, continuously compounded interest is 5%. Use this data to compute price and delta of the call basket option with the Ju approximation model.

```
Settle = 'May-1-2009';
Maturity  = 'May-1-2010';

% Define RateSpec
Rate = 0.05;
Compounding = -1;
RateSpec = intenvset('ValuationDate', Settle, 'StartDates', ...
Settle, 'EndDates', Maturity, 'Rates', Rate, 'Compounding', Compounding);

% Define the Correlation matrix. Correlation matrices are symmetric, and
% have ones along the main diagonal.
NumInst  = 5;
InstIdx = ones(NumInst,1);
Corr = diag(ones(5,1), 0);

% Define BasketStockSpec
AssetPrice =  [110; 75; 40; 125; 92];
Volatility = 0.2;
Quantity = [0.05; 0.15; 0.2; 0.25; 0.35];
BasketStockSpec = basketstockspec(Volatility, AssetPrice, Quantity, Corr);
```

```
% Compute the price of the call basket option. Calculate also the delta
% of the first stock.
OptSpec = {'call'};
Strike = 90;
OutSpec = {'Price','Delta'};
UndIdx = 1; % First element in the basket
[Price, Delta] = basketsensbyju(RateSpec, BasketStockSpec, OptSpec, Strike, Settle, ...
Maturity, 'OutSpec', OutSpec,'UndIdx', UndIdx)
```

This returns:

```
Price =

   5.16098

Delta =

   0.02972
```

Compute `Delta` with respect to the second asset:

```
UndIdx = 2; % Second element in the basket
OutSpec = {'Delta'};
Delta = basketsensbyju(RateSpec, BasketStockSpec, OptSpec, Strike, Settle, Maturity, ...
'OutSpec',OutSpec,'UndIdx',UndIdx)

Delta =

   0.09063
```

References

Nengjiu Ju, “Pricing Asian and Basket Options Via Taylor Expansion”, *Journal of Computational Finance*, Vol. 5, 2002.

See Also

`basketstockspec` | `basketbyju`

How To

- “Basket Option” on page 3-25

Purpose

Calculate price and sensitivities for basket options using Longstaff-Schwartz model

Syntax

```
PriceSens = basketsensbyls(RateSpec, BasketStockSpec, OptSpec,
   Strike,
Settle, ExerciseDates)
PriceSens = basketsensbyls(RateSpec, BasketStockSpec, OptSpec,
   Strike,
Settle, ExerciseDates, 'ParameterName', ParameterValue ...)
```

Description

`PriceSens = basketsensbyls(RateSpec, BasketStockSpec, OptSpec, Strike, Settle, ExerciseDates)` prices basket options using the Longstaff-Schwartz model.

`PriceSens = basketsensbyls(RateSpec, BasketStockSpec, OptSpec, Strike, Settle, ExerciseDates, 'ParameterName', ParameterValue ...)` accepts optional inputs as one or more comma-separated parameter/value pairs. *'ParameterName'* is the name of the parameter inside single quotes. *'ParameterValue* is the value corresponding to *'ParameterName'*. Specify parameter-value pairs in any order. Names are case-insensitive and partial string matches are allowable, if no ambiguities exist.

Input Arguments

`RateSpec`

Annualized, continuously compounded rate term structure. For more information on the interest rate specification, see `intenvset`.

`BasketStockSpec`

`BasketStock` specification. For information on the basket of stocks specification, see `basketstockspec`.

`OptSpec`

String or 2-by-1 cell array of the strings `'call'` or `'put'`.

`Strike`

The option strike price:

- For a European or Bermuda option, `Strike` is a scalar (European) or `1`-by-`NSTRIKES` (Bermuda) vector of strike price.
- For an American option, `Strike` is a scalar vector of strike price.

`Settle`

Scalar of settlement or trade date.

`ExerciseDates`

The exercise date for the option:

- For a European or Bermuda option, `ExerciseDates` is a `1`-by-`1` (European) or `1`-by-`NSTRIKES` (Bermuda) vector of exercise dates. For a European option, there is only one `ExerciseDate` on the option expiry date.
- For an American option, `ExerciseDates` is a `1`-by-`2` vector of exercise date boundaries. The option exercises on any date between or including the pair of dates on that row. If there is only one non-`NaN` date, or if `ExerciseDates` is `1`-by-`1`, the option exercises between the `Settle` date and the single listed `ExerciseDate`.

Parameter–Value Pairs

`AmericanOpt`

Parameter values are a scalar flag.

- 0 — European/Bermuda
- 1 — American

Default: 0

`NumPeriods`

Parameter value is a scalar number of simulation periods. `NumPeriods` is considered only when pricing European basket options. For American and Bermuda basket options, `NumPeriod` equals the number of exercise days during the life of the option.

Default: 100

`NumTrials`

Parameter value is a scalar number of independent sample paths (simulation trials).

Default: 1000

`OutSpec`

Parameter value is an `NOUT`-by-`1` or `1`-by-`NOUT` cell array of strings indicating the nature and order of the outputs for the function. Possible values are `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, and `'All'`. For example, `OutSpec = {'Price', 'Lamba', 'Rho'}` specifies that the output is `Price`, `Lambda`, and `Rho`, in that order.

`OutSpec = {'All'}` specifies that the output should be `Delta`, `Gamma`, `Vega`, `Lambda`, `Rho`, `Theta`, and `Price`, in that order. This is the same as specifying `OutSpec` as `OutSpec = {'Delta', 'Gamma', 'Vega', 'Lambda', 'Rho', 'Theta', 'Price'};`.

Default: `OutSpec = {'Price'}`

`UndIdx`

Scalar of the indice of the underlying instrument to compute the sensitivity.

Default: `UndIdx = []`

Output Arguments

`PriceSens`

Expected prices or sensitivities values.

Examples

Find a European put basket option of two stocks. The basket contains 50% of each stock. The stocks are currently trading at $90 and $75, with annual volatilities of 15%. Assume that the correlation between the assets is zero. On May 1, 2009, an investor wants to buy a one-year put option with a strike price of $80. The current annualized, continuously compounded interest is 5%. Use this data to compute price and delta of the put basket option with the Longstaff-Schwartz approximation model.

```
Settle = 'May-1-2009';
Maturity  = 'May-1-2010';

% Define RateSpec
Rate = 0.05;
Compounding = -1;
RateSpec = intenvset('ValuationDate', Settle, 'StartDates',...
Settle, 'EndDates', Maturity, 'Rates', Rate, 'Compounding', Compounding);

% Define the Correlation matrix. Correlation matrices are symmetric,
% and have ones along the main diagonal.
NumInst  = 2;
InstIdx = ones(NumInst,1);
Corr = diag(ones(NumInst,1), 0);

% Define BasketStockSpec
AssetPrice =  [90; 75];
Volatility = 0.15;
Quantity = [0.50; 0.50];
BasketStockSpec = basketstockspec(Volatility, AssetPrice, Quantity, Corr);

% Compute the price of the put basket option. Calculate also the delta
% of the first stock.
OptSpec = {'put'};
```

```
Strike = 80;
OutSpec = {'Price','Delta'};
UndIdx = 1; % First element in the basket

[PriceSens, Delta] = basketsensbyls(RateSpec, BasketStockSpec, OptSpec,...
Strike, Settle, Maturity,'OutSpec', OutSpec,'UndIdx', UndIdx)
```

This returns:

```
PriceSens =

    1.08519

Delta =

   -0.10311
```

Compute the `Price` and `Delta` of the basket with a correlation of -20%:

```
NewCorr = [1 -0.20; -0.20 1];

% Define the new BasketStockSpec.
BasketStockSpec = basketstockspec(Volatility, AssetPrice, Quantity, NewCorr);

% Compute the price and delta of the put basket option.
[PriceSens, Delta] = basketsensbyls(RateSpec, BasketStockSpec, OptSpec,...
Strike, Settle, Maturity,'OutSpec', OutSpec,'UndIdx', UndIdx)

PriceSens =

    0.83903

Delta =

   -0.08847
```

References

Longstaff, F.A., and E.S. Schwartz, "Valuing American Options by Simulation: A Simple Least-Squares Approach", *The Review of Financial Studies*,Vol. 14, No. 1, Spring 2001, pp. 113–147.

See Also

`basketstockspec` | `basketbyls`

How To

- "Basket Option" on page 3-25

Purpose

Specify basket stock structure

Syntax

```
BasketStockSpec = basketstockspec(Sigma, AssetPrice, Quantity,
Correlation)
BasketStockSpec = basketstockspec(Sigma, AssetPrice, Quantity,
Correlation, 'ParameterName',ParameterValue ...)
```

Description

`BasketStockSpec = basketstockspec(Sigma, AssetPrice, Quantity, Correlation)` creates a basket stock structure.

`BasketStockSpec = basketstockspec(Sigma, AssetPrice, Quantity, Correlation, 'ParameterName',ParameterValue ...)` accepts optional inputs as one or more comma-separated parameter/value pairs. `'ParameterName'` is the name of the parameter inside single quotes. `'ParameterValue` is the value corresponding to `'ParameterName'`. Specify parameter-value pairs in any order. Names are case-insensitive and partial string matches are allowable, if no ambiguities exist.

Input Arguments

`Sigma`

NINST-by-1 vector of decimal annual price volatility of the underlying security.

`AssetPrice`

NINST-by-1 vector of underlying asset price values at time 0.

`Quantity`

NINST-by-1 vector of quantities of the instruments contained in the basket.

`Correlation`

NINST-by-NINST matrix of correlation values.

Parameter–Value Pairs

`DividendAmounts`

NINST-by-1 cell array specifying the dividend amounts for basket instruments. Each element of the cell array is a 1-by-NDIV row vector of cash dividends or a scalar representing a continuous annualized dividend yield for the corresponding instrument.

`DividendType`

NINST-by-1 cell array of strings specifying each stock's dividend type. Dividend type must be either `cash` for actual dollar dividends or `continuous` for continuous dividend yield. .

`ExDividendDates`

NINST-by-1 cell array specifying the ex-dividend dates for the basket instruments. Each row is a 1-by-NDIV matrix of ex-dividend dates for `cash` type. For rows that correspond to basket instruments with `continuous` dividend type, the cell is empty. If none of the basket instruments pay `continuous` dividends, do not specify `ExDividendDates`.

Output Arguments

`BasketStockSpec`

Structure encapsulating the properties of a basket stock structure.

Examples

Find a basket option of three stocks. The stocks are currently trading at $56, $92 and $125 with annual volatilities of 20%, 12% and 15%, respectively. The basket option contains 25% of the first stock, 40% of the second stock, and 35% of the third. The first stock provides a continuous dividend of 1%, while the other two provide no dividends. The correlation between the first and second asset is 30%, between the second and third asset 11%, and between the first and third asset 16%. Use this data to create the `BasketStockSpec` structure:

```
AssetPrice = [56;92;125];
Sigma = [0.20;0.12;0.15];

% Create the Correlation matrix. Correlation matrices are symmetric and
% have ones along the main diagonal.
NumInst  = 3;
```

```
Corr = zeros(NumInst,1);
Corr(1,2) = .30;
Corr(2,3) = .11;
Corr(1,3) = .16;
Corr = triu(Corr,1) + tril(Corr',-1) + diag(ones(NumInst,1), 0);

% Define dividends
DivType = cell(NumInst,1);
DivType{1}='continuous';
DivAmounts = cell(NumInst,1);
DivAmounts{1} = 0.01;

Quantity = [0.25; 0.40; 0.35];

BasketStockSpec = basketstockspec(Sigma, AssetPrice, Quantity, Corr, ...
'DividendType', DivType, 'DividendAmounts', DivAmounts)
```

This returns:

```
BasketStockSpec =

             FinObj: 'BasketStockSpec'
              Sigma: [3x1 double]
         AssetPrice: [3x1 double]
           Quantity: [3x1 double]
        Correlation: [3x3 double]
       DividendType: {3x1 cell}
    DividendAmounts: {3x1 cell}
    ExDividendDates: {3x1 cell}
```

Examine the `BasketStockSpec` structure:

```
>>BasketStockSpec.Correlation
ans =

    1.0000    0.3000    0.1600
    0.3000    1.0000    0.1100
```

```
    0.1600    0.1100    1.0000
```

Find a basket option of two stocks. The stocks are currently trading at $60 and $55 with volatilities of 30% per annum. The basket option contains 50% of each stock. The first stock provides a cash dividend of $0.25 on May 1, 2009 and September 1, 2009. The second stock provides a continuous dividend of 3%. The correlation between the assets is 40%. Use this data to create the structure `BasketStockSpec`:

```
AssetPrice = [60;55];
Sigma = [0.30;0.30];

% Create the Correlation matrix. Correlation matrices are symmetric and
% have ones along the main diagonal.
Correlation = [1 0.40;0.40 1];

% Define dividends
NumInst  = 2;
DivType = cell(NumInst,1);
DivType{1}='cash';
DivType{2}='continuous';

DivAmounts = cell(NumInst,1);
DivAmounts{1} = [0.25 0.25];
DivAmounts{2} = 0.03;

ExDates = cell(NumInst,1);
ExDates{1} = {'May-1-2009' 'Sept-1-2009'};

Quantity = [0.5; 0.50];

BasketStockSpec = basketstockspec(Sigma, AssetPrice, Quantity, Correlation, ...
'DividendType', DivType, 'DividendAmounts', DivAmounts, 'ExDividendDates',ExDates)
```

This returns:

```
BasketStockSpec =

             FinObj: 'BasketStockSpec'
              Sigma: [2x1 double]
         AssetPrice: [2x1 double]
           Quantity: [2x1 double]
        Correlation: [2x2 double]
       DividendType: {2x1 cell}
    DividendAmounts: {2x1 cell}
    ExDividendDates: {2x1 cell}
```

Examine the `BasketStockSpec` structure:

```
>>BasketStockSpec.DividendType

ans =

    'cash'
    'continuous'
```

See Also

`basketbyls` | `basketbyju` | `basketsensbyju` | `basketsensbyls` | `stockspec` | `intenvset`

How To

- "Basket Option" on page 3-25

bdtprice

Purpose

Instrument prices from BDT interest-rate tree

Syntax

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

Arguments

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

Description

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

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

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

`PriceTree.PTree` contains the clean prices.

`PriceTree.AITree` contains the accrued interest.

`PriceTree.tObs` contains the observation times.

`bdtprice` handles instrument types: `'Bond'`, `'CashFlow'`, `'OptBond'`, `'OptEmBond'`, `'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.
- `optembndbybdt`: Price a bond with embedded option by a BDT tree.
- `swapbybdt`: Price a swap from a BDT tree.
- `swaptionbybdt`: Price a swaption from a BDT tree.

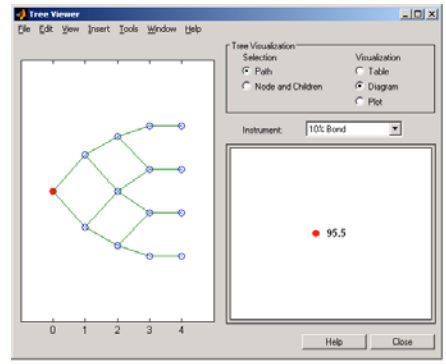
Examples

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

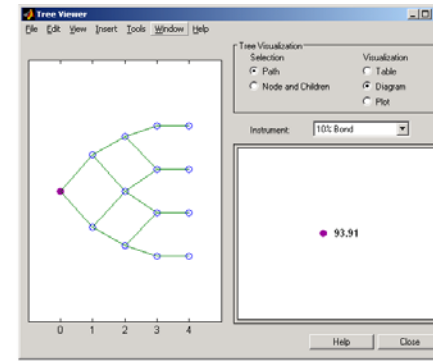
```
load deriv.mat;
BDTSubSet = instselect(BDTInstSet,'Type', {'Bond', 'Cap'});

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 =

   95.5030
   93.9079
    1.4863
```

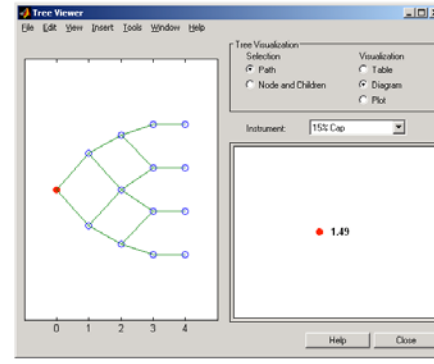
You can use `treeviewer` to see the prices of these three instruments along the price tree.



First 10% Bond (Maturity 2003)



Second 10% Bond (Maturity 2004)



15% Cap

See Also

`bdtsens`, `bdttree`, `instadd`, `intenvprice`, `intenvsens`

bdtsens

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'`, `'OptEmBond'`, `'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% Bo
nd
2     Bond 0.1        01-Jan-2000   01-Jan-2004  2       10% Bo
nd

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

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

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

```
Delta =

 -232.6681
 -281.0517
   78.3776

Gamma =

  1.0e+003 *

    0.8037
    1.1819
    0.7490
```

See Also

bdtprice, bdttree, bdtvolspec, instadd

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 1 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.

bdttimespec

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`

Purpose

Construct BDT interest-rate tree

Syntax

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

Arguments

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

Description

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

Examples

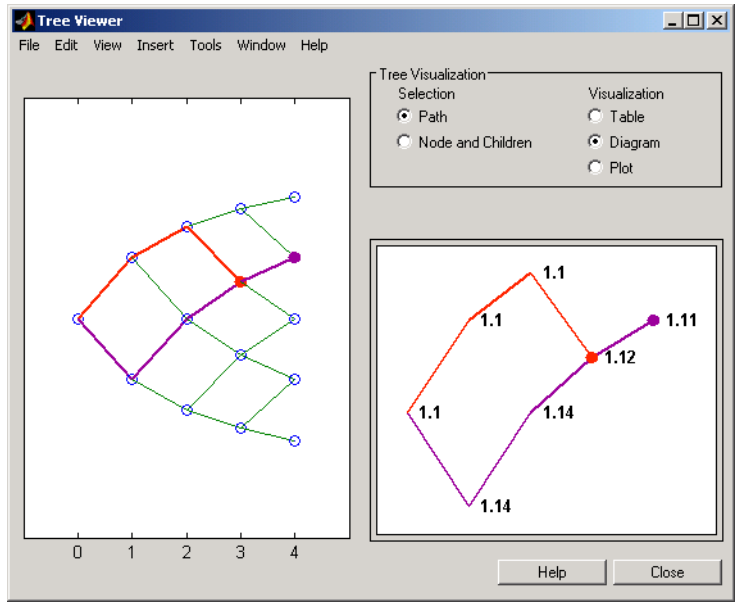
Using the data provided, create a BDT volatility specification (`VolSpec`), rate specification (`RateSpec`), and tree time layout specification (`TimeSpec`). Then use these specifications to create a BDT tree with `bdttree`.

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

RateSpec = intenvset('Compounding', Compounding,...
```

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

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

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

```
treeviewer(BDTTree)
```



See Also

`bdtprice`, `bdttimespec`, `bdtvolspec`, `intenvset`

Purpose

Specify BDT interest-rate volatility process

Syntax

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

Arguments

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

Description

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

Examples

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

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

BDTVolSpec = bdtvolspec(ValuationDate, EndDates, Volatility)

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

```
                VolInterpMethod: 'linear'
```

See Also

bdttree, interp1

Purpose

Instrument prices from Black-Karasinski interest-rate tree

Syntax

```
[Price, PriceTree] = bkprice(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

`[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'`, `'OptEmBond'`, `'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.
- `optembndbybk`: Price a bond with embedded option by a Black-Karasinski tree.
- `swapbybk`: Price a swap from a Black-Karasinski tree.
- `swaptionbybk`: Price a swaption from a Black-Karasinski tree.

Examples

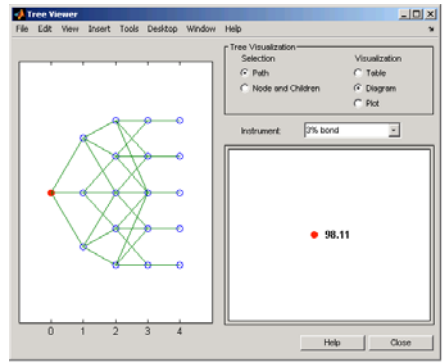
Load the BK tree and instruments from the data file `deriv.mat`. Price the cap and bond instruments contained in the instrument set.

```
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);
```

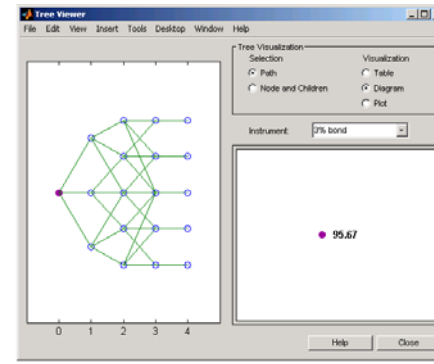
```
Price =

   98.1096
   95.6734
    2.2706
```

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

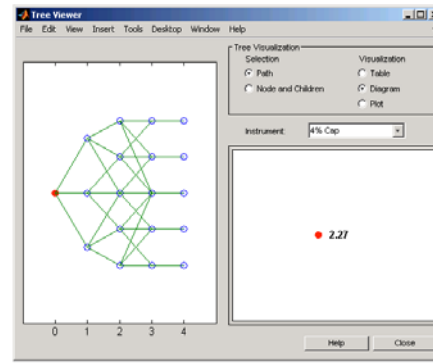
```
treeviewer(PriceTree, BKSubSet)
```



First 3% Bond (Maturity 2007)



Second 3% Bond (Maturity 2008)



4% Cap

See Also

`bksens`, `bktree`, `instadd`, `intenvprice`, `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'`, `'OptEmBond'`, `'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% Bon
d
2     Bond 0.03       01-Jan-2004   01-Jan-2008  1       3% Bon
d

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

[Delta, 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 1 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
     EndMonthRule: 1
```

See Also

`bktree`, `bkvolspec`, `hwtree`

bktree

Purpose

Construct Black-Karasinski interest-rate tree

Syntax

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

Arguments

`VolSpec`	Volatility process specification. See `bkvolspec` 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 BK tree and the `Compounding` rule for date to time mapping and price-yield formulas. See `bktimespec` 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

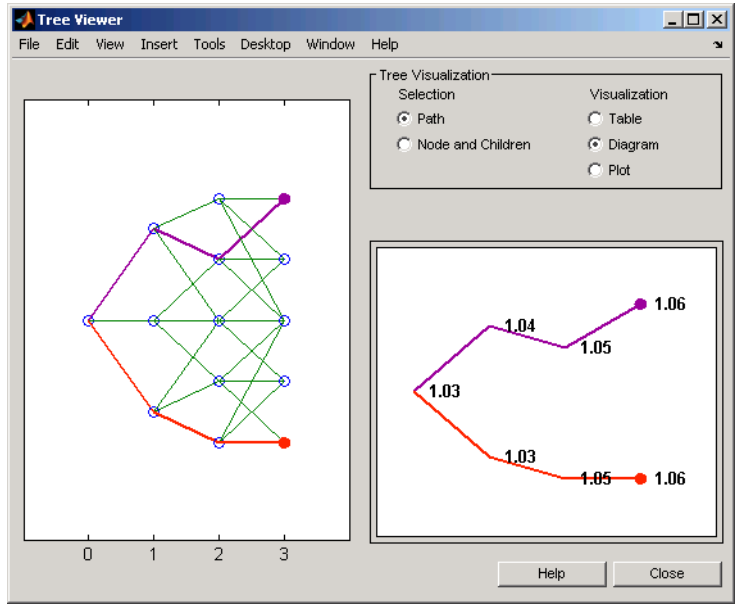
Using the data provided, create a BK volatility specification (`VolSpec`), rate specification (`RateSpec`), and tree time layout specification (`TimeSpec`). Then use these specifications to create a BK tree using `bktree`.

```
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 `treeviewer` to observe the tree you have created.

```
treeviewer(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 `interp1` 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(BDTTree, CouponRate, Settle,
Maturity, Period, Basis, EndMonthRule, IssueDate,
FirstCouponDate,  LastCouponDate, StartDate, Face, Options)
```

Arguments

`BDTTree`	Interest-rate tree structure created by `bdttree`.
`CouponRate`	Decimal annual rate.
`Settle`	Settlement date. A vector of serial date numbers or date strings. `Settle` must be earlier than `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. • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese)

	• 8 = actual/actual (ICMA) • 9 = actual/360 (ICMA) • 10 = actual/365 (ICMA) • 11 = 30/360E (ICMA) • 12 = actual/actual (ISDA) • 13 = BUS/252
`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`	(Optional) Date when a bond actually starts (the date from which a bond cash flow is considered). To make an instrument forward-starting, specify this date as a future date. If you do not specify `StartDate`, the effective start date is the `Settle` date.
`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 BDT tree. The bond argument `Settle` is ignored.

Description

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

Price =

  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 `bktree`.
`CouponRate`	Decimal annual rate.
`Settle`	Settlement date. A vector of serial date numbers or date strings. `Settle` must be earlier than `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. • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese)

	• 8 = actual/actual (ICMA) • 9 = actual/360 (ICMA) • 10 = actual/365 (ICMA) • 11 = 30/360E (ICMA) • 12 = actual/actual (ISDA) • 13 = BUS/252
`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 before 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`	(Optional) Date when a bond actually starts (the date from which a bond cash flow is considered). To make an instrument forward-starting, specify this date as a future date. If you do not specify `StartDate`, the effective start date is the `Settle` date.
`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 `hjmtree`.
`CouponRate`	Decimal annual rate.
`Settle`	Settlement date. A vector of serial date numbers or date strings. `Settle` must be earlier than `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. • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese)

- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

`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 before 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`	(Optional) Date when a bond actually starts (the date from which a bond cash flow is considered). To make an instrument forward-starting, specify this date as a future date. If you do not specify `StartDate`, the effective start date is the `Settle` date.
`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 `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. • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese)

	• 8 = actual/actual (ICMA) • 9 = actual/360 (ICMA) • 10 = actual/365 (ICMA) • 11 = 30/360E (ICMA) • 12 = actual/actual (ISDA) • 13 = BUS/252
`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 before 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`	(Optional) Date when a bond actually starts (the date from which a bond cash flow is considered). To make an instrument forward-starting, specify this date as a future date. If you do not specify `StartDate`, the effective start date is the `Settle` date.
`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] = 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 `intenvset` for information on creating `RateSpec`.
`CouponRate`	Decimal annual rate.
`Settle`	Settlement date. A vector of serial date numbers or date strings. `Settle` must be earlier than `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. • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese)

	• 8 = actual/actual (ICMA) • 9 = actual/360 (ICMA) • 10 = actual/365 (ICMA) • 11 = 30/360E (ICMA) • 12 = actual/actual (ISDA) • 13 = BUS/252
`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 before 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`	(Optional) Date when a bond actually starts (the date from which a bond cash flow is considered). To make an instrument forward-starting, specify this date as a future date. If you do not specify `StartDate`, the effective start date is the `Settle` date.
`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

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 1, the second-to-top is 2, 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])
```

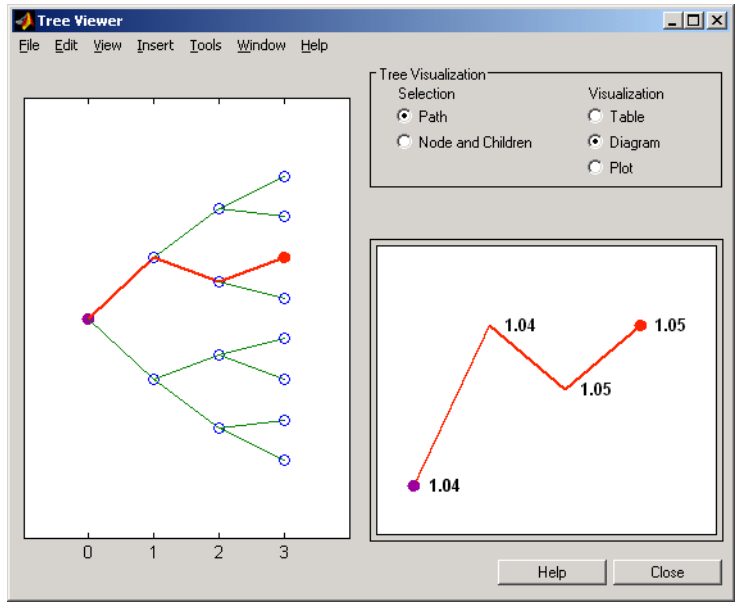
returns the rates at the tree nodes located by taking the up branch, then the down branch, and finally the up branch again.

```
FwdRates =

    1.0356
    1.0364
    1.0526
```

```
    1.0463
```

You can visualize this with the `treeviewer` function.

```
treeviewer(HJMTree)
```



See Also

`bushshape`, `mkbush`

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.

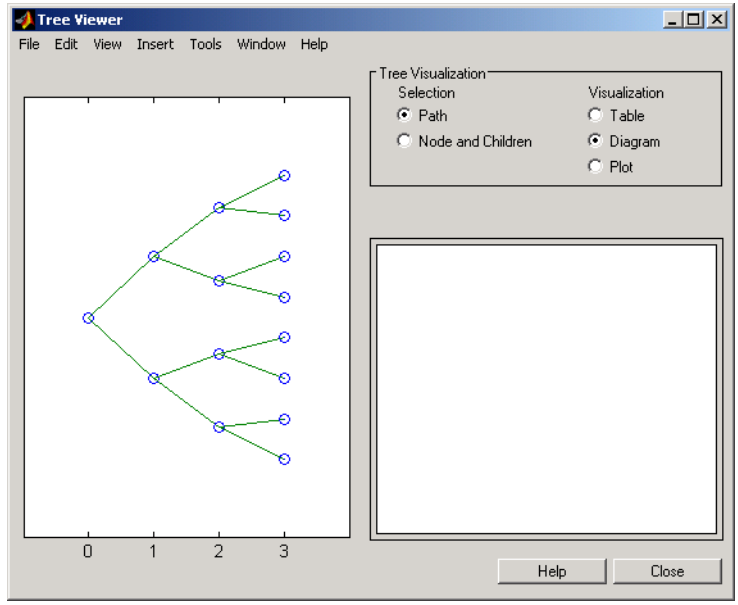
`Trim` is `1` if `NumPos` decreases by 1 when moving from one time level to the next. Otherwise, it is `0`.

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
```

```
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(BDTTree, Strike, Settle,
Maturity, Reset, Basis, Principal, Options)
```

Arguments

`BDTTree`	Interest-rate tree structure created by `bdttree`.
`Strike`	Number of instruments (`NINST`)-by-`1` vector of rates at which the cap is exercised.
`Settle`	Settlement dates. `NINST`-by-`1` vector of dates representing the settlement dates of the cap.
`Maturity`	`NINST`-by-`1` vector of dates representing the maturity dates of the cap.
`Reset`	(Optional) `NINST`-by-`1` vector representing the frequency of payments per year. Default = `1`.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers.

- 0 = actual/actual (default)
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)

- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

`Principal`	(Optional) The notional principal amount. Default = `100`.
`Options`	(Optional) Derivatives pricing options structure created with `derivset`.

Description

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

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

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

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

Examples

Example 1. Price a 3% cap instrument using a BDT interest-rate tree.

Load the file `deriv.mat`, which provides `BDTTree`. The `BDTTree` 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(BDTTree, 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.

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

Price =

    1.6923
```

See Also

bdttree, cfbybdt, floorbybdt, swapbybdt

capbybk

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.

- 0 = actual/actual (default)
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)

- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

`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 caps using Black option pricing model

Syntax

```
[CapPrice, Caplets] = capbyblk(RateSpec, Strike, Settle,
Maturity, Volatility)
[CapPrice, Caplets] = capbyblk(RateSpec, Strike, Settle,
Maturity, Volatility, 'Name1', Value1...)
```

Arguments

`RateSpec`	The annualized, continuously compounded rate term structure. For more information, see `intenvset`.
`Strike`	`NINST`-by-`1` vector of rates at which the cap is exercised, as a decimal number.
`Settle`	Scalar representing the settle date of the cap.
`Maturity`	Scalar representing the maturity date of the cap.
`Volatility`	`NINST`-by-`1` vector of volatilities.
`Reset`	(Optional) `NINST`-by-`1` vector representing the frequency of payments per year. Default is `1`.
`Principal`	(Optional) `NINST`-by-`1` vector representing the notional principal amount. Default is `100`.
`Basis`	`NINST`-by-`1` vector representing the basis used when annualizing the input forward rate. • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European)

- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

`ValuationDate` (Optional) Scalar representing the observation date of the investment horizons. The default is the `Settle` date.

Note All optional inputs are specified as matching parameter name/value pairs. The parameter name is specified as a character string, followed by the corresponding parameter value. You can specify parameter name/value pairs in any order. Names are case-insensitive and partial string matches are allowed provided no ambiguities exist.

Description

```
[CapPrice, Caplets] = capbyblk(RateSpec, Strike, Settle,
Maturity, Volatility)
```

```
[CapPrice, Caplets] = capbyblk(RateSpec, Strike, Settle,
Maturity, Volatility, 'Name1', Value1...)
```

Use `capbyblk` to price caps using the Black option pricing model.

The outputs are:

- `CapPrice` — `NINST`-by-`1` expected prices of the cap.
- `Caplets` — `NINST`-by-`NCF` array of caplets, padded with `NaN`s.

Examples

Consider an investor who gets into a contract that caps the interest rate on a $100,000 loan at 8% quarterly compounded for 3 months, starting on January 1, 2009. Assuming that on January 1, 2008 the zero rate is 6.9394% continuously compounded and the volatility is 20%, use this data to compute the cap price.

Calculate the `RateSpec`:

```
ValuationDate = 'Jan-01-2008';
EndDates ='April-01-2010';
Rates = 0.069394;
Compounding = -1;
Basis = 1;

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

Compute the price of the cap:

```
Settle = 'Jan-01-2009'; % cap starts in a year
Maturity = 'April-01-2009';
Volatility = 0.20;
CapRate = 0.08;
CapReset = 4;
Principal=100000;

CapPrice = capbyblk(RateSpec,  CapRate, Settle, Maturity, Volatility,...
'Reset',CapReset,'ValuationDate',ValuationDate,'Principal', Principal,...
'Basis', Basis)

CapPrice =

   51.6125
```

See Also

`floorbyblk`

capbyhjm

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 `hjmtree`.
`Strike`	Number of instruments (`NINST`)-by-`1` vector of rates at which the cap is exercised.
`Settle`	Settlement dates. `NINST`-by-`1` vector of dates representing the settlement dates of the cap.
`Maturity`	`NINST`-by-`1` vector of dates representing the maturity dates of the cap.
`Reset`	(Optional) `NINST`-by-`1` vector representing the frequency of payments per year. Default = `1`.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers.

- 0 = actual/actual (default)
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)

- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

`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 =

  6.2831
```

See Also

cfbyhjm, floorbyhjm, hjmtree, swapbyhjm

Purpose

Price cap instrument from Hull-White interest-rate tree

Syntax

```
[Price, PriceTree] = capbyhw(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 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.

- 0 = actual/actual (default)
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)

- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

`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

cashbybls

Purpose

Calculate price of cash-or-nothing digital options using Black-Scholes model

Syntax

```
Price = cashbybls(RateSpec, StockSpec, Settle, Maturity,
OptSpec, Strike, Payoff)
```

Arguments

`RateSpec`	The annualized, continuously compounded rate term structure. For information on the interest rate specification, see `intenvset`.
`StockSpec`	Stock specification. See `stockspec`.
`Settle`	`NINST`-by-`1` vector of settlement or trade dates.
`Maturity`	`NINST`-by-`1` vector of maturity dates.
`OptSpec`	`NINST`-by-`1` cell array of strings `'call'` or `'put'`.
`Strike`	`NINST`-by-`1` vector of strike price values.
`Payoff`	`NINST`-by-`1` vector of payoff values or the amount to be paid at expiration.

Description

`Price = cashbybls(RateSpec, StockSpec, Settle, Maturity, OptSpec, Strike, Payoff)` computes cash-or-nothing option prices using the Black-Scholes option pricing model.

`Price` is a `NINST`-by-`1` vector of expected option prices.

Examples

Consider a European call and put cash-or-nothing options on a futures contract with and exercise strike price of $90, a fixed payoff of $10 that expires on October 1, 2008. Assume that on January 1, 2008, the contract trades at $110, and has a volatility of 25% per annum and the risk-free rate is 4.5% per annum. Using this data, calculate the price of the call and put cash-or-nothing options on the futures contract.

Create the `RateSpec`:

```
Settle = 'Jan-1-2008';
Maturity = 'Oct-1-2008';
Rates = 0.045;
Compounding = -1;
Basis = 1;
RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle,...
'EndDates', Maturity, 'Rates', Rates, 'Compounding', Compounding, 'Basis', Basis);
```

Define the `StockSpec`:

```
AssetPrice = 110;
Sigma = .25;
DivType = 'Continuous';
DivAmount = Rates;
StockSpec = stockspec(Sigma, AssetPrice, DivType, DivAmount);
```

Define the call and put options:

```
OptSpec = {'call'; 'put'};
Strike = 90;
Payoff = 10;
```

Calculate the price:

```
Pcon = cashbybls(RateSpec, StockSpec, Settle,...
Maturity, OptSpec, Strike, Payoff)

Pcon =

    7.6716
    1.9965
```

See Also

`assetbybls`, `cashsensbybls`, `gapbybls`, `supersharebybls`

cashsensbybls

Purpose

Calculate price and sensitivities of cash-or-nothing digital options using Black-Scholes model

Syntax

```
PriceSens = cashsensbybls(RateSpec, StockSpec, Settle,
Maturity, OptSpec, Strike, Payoff)
PriceSens = cashsensbybls(RateSpec, StockSpec, Settle,
Maturity, OptSpec, Strike, Payoff, OutSpec)
```

Arguments

`RateSpec`	The annualized, continuously compounded rate term structure. For information on the interest rate specification, see `intenvset`.
`StockSpec`	Stock specification. See `stockspec`.
`Settle`	`NINST`-by-`1` vector of settlement or trade dates.
`Maturity`	`NINST`-by-`1` vector of maturity dates.
`OptSpec`	`NINST`-by-`1` cell array of strings `'call'` or `'put'`.
`Strike`	`NINST`-by-`1` vector of strike price values.
`Payoff`	`NINST`-by-`1` vector of payoff values or the amount to be paid at expiration.
`OutSpec`	(Optional) All optional inputs are specified as matching parameter name/value pairs. The parameter name is specified as a character string, followed by the corresponding parameter value. You can specify parameter name/value pairs in any order. Names are case-insensitive and partial string matches are allowed provided no ambiguities exist. Valid parameter names are: • `NOUT`-by-`1` or `1`-by-`NOUT` cell array of strings indicating the nature and order of the outputs for the function. Possible values are `'Price'`,

'Delta', 'Gamma', 'Vega', 'Lambda', 'Rho', 'Theta', or 'All'.

For example, OutSpec = {'Price'; 'Lamba'; 'Rho'} specifies that the output should be Price, Lambda, and Rho, in that order.

To invoke from a function: [Price, Lambda, Rho] = cashsensbybls(..., 'OutSpec', {'Price', 'Lamba', 'Rho'})

OutSpec = {'All'} specifies that the output should be Delta, Gamma, Vega, Lambda, Rho, Theta, and Price, in that order. This is the same as specifying OutSpec as OutSpec = {'Delta', 'Gamma', 'Vega', 'Lambda', 'Rho', 'Theta', 'Price'};.

- Default is OutSpec = {'Price'}.

Description

PriceSens = cashsensbybls(RateSpec, StockSpec, Settle, Maturity, OptSpec, Strike, Payoff) computes cash-or-nothing option prices using the Black-Scholes option pricing model.

PriceSens = cashsensbybls(RateSpec, StockSpec, Settle, Maturity, OptSpec, Strike, Payoff, OutSpec) includes an OutSpec argument defined as parameter/value pairs, and computes cash-or-nothing option prices and sensitivities using the Black-Scholes option pricing model.

PriceSens is a NINST-by-1 vector of expected option prices and sensitivities.

Examples

Consider a European call and put cash-or-nothing options on a futures contract with an exercise price of $90, and a fixed payoff of $10 that expires on January 1, 2009. Assume that on October 1, 2008 the contract trades at $110, and has a volatility of 25% per annum and the risk-free rate is 4.5% per annum. Using this data, calculate the

price and sensitivity of the call and put cash-or-nothing options on the futures contract.

Create the `RateSpec`:

```
Settle = 'Jan-1-2008';
Maturity = 'Oct-1-2008';
Rates = 0.045;
Compounding = -1;
Basis = 1;
RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle,...
'EndDates', Maturity, 'Rates', Rates, 'Compounding', Compounding, 'Basis', Basis);
```

Define the `StockSpec`:

```
AssetPrice = 110;
Sigma = .25;
DivType = 'Continuous';
DivAmount = Rates;
StockSpec = stockspec(Sigma, AssetPrice, DivType, DivAmount);
```

Define the call and put options:

```
OptSpec = {'call'; 'put'};
Strike = 90;
Payoff = 10;
```

Compute the gamma, theta, and price:

```
OutSpec = { 'gamma';'theta';'price'};
[Gamma, Theta, Price] = cashsensbybls(RateSpec, StockSpec,...
Settle, Maturity, OptSpec, Strike, Payoff, 'OutSpec', OutSpec)

Gamma =

   -0.0050
    0.0050
```

```
Theta =

   -2.2489
    1.8139


Price =

    7.6716
    1.9965
```

See Also

cashbybls

cfbybdt

Purpose

Price cash flows from BDT interest-rate tree

Syntax

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

Arguments

`BDTTree`	Forward-rate tree structure created by `bdttree`.
`CFlowAmounts`	Number of instruments (`NINST`) by maximum number of cash flows (`MOSTCFS`) matrix of cash flow amounts. Each row is a list of cash flow values for one instrument. If an instrument has fewer than `MOSTCFS` cash flows, the end of the row is padded with `NaN`s.
`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 BDT tree. The cash flow argument, `Settle`, is ignored.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European)

- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

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

Description

`[Price, PriceTree] = cfbybdt(BDTTree, CFlowAmounts, CFlowDates, Settle, Basis, Options)` prices cash flows from a BDT interest-rate tree.

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

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

Examples

Price a portfolio containing two cash flow instruments paying interest annually over the four-year period from January 1, 2000 to January 1, 2004.

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

```
load deriv.mat;
```

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

```
BDTTree.RateSpec.ValuationDate
```

```
ans =

      730486
```

Provide values for the other required arguments.

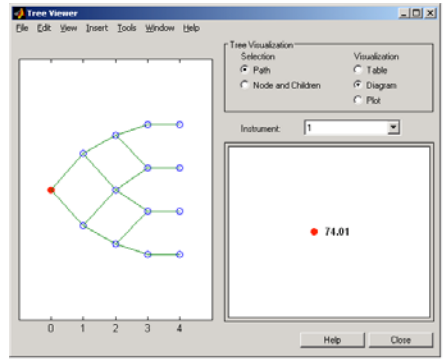
```
CFlowAmounts =[5 NaN 5.5 105; 5 0 6 105];
CFlowDates = [730852, NaN, 731582, 731947;
              730852, 731217, 731582, 731947];
```

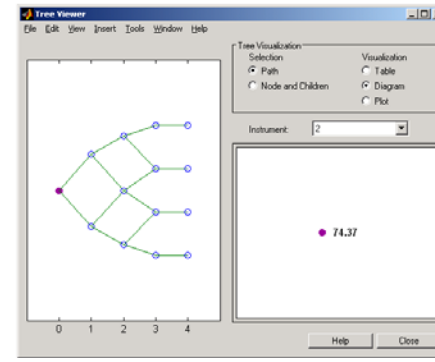
Use this information to compute the prices of the two cash flow instruments.

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

Price =

   74.0112
   74.3671

PriceTree =

    FinObj: 'BDTPriceTree'
      tObs: [0 1.00 2.00 3.00 4.00]
     PTree: {1x5 cell}
```

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

```
treeviewer(PriceTree)
```





See Also

`bdttree`, `bdtprice`, `cfamounts`, `instcf`

cfbybk

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 `NaN`s.
`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. • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European)

- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

`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.

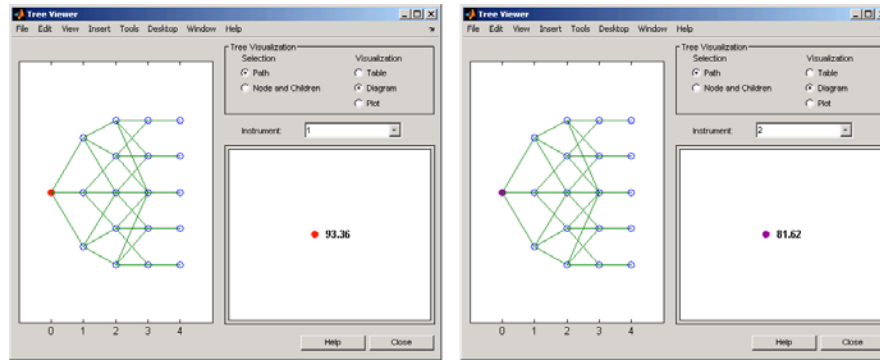
```
CFlowAmounts =[5 NaN 5.5 105; 5 0 6 105];
CFlowDates = [732678, NaN, 733408,733774;
              732678, 733034, 733408, 734774];
```

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`

cfbyhjm

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 `NaN`s.
`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. • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European)

- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

`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.

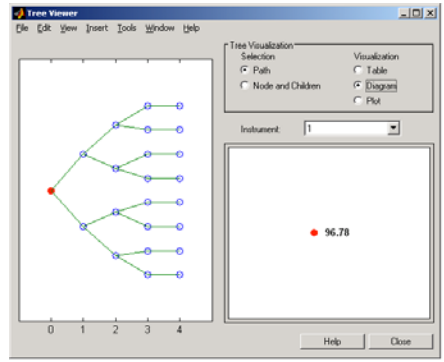
```
CFlowAmounts =[5 NaN 5.5 105; 5 0 6 105];
CFlowDates = [730852, NaN, 731582, 731947;
              730852, 731217, 731582, 731947];
```

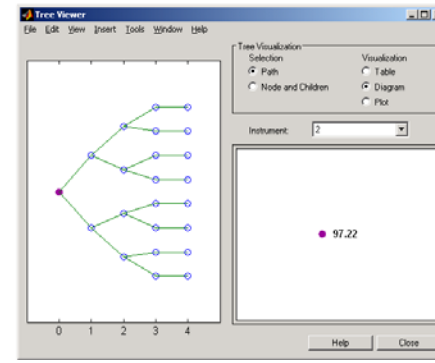
Use this information to compute the prices of the two cash flow instruments.

```
[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`

cfbyhw

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 `hwtree`.
`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 `NaN`s.
`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. • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European)

- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

`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.

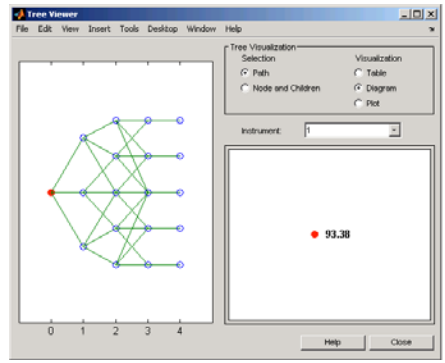
```
CFlowAmounts =[5 NaN 5.5 105; 5 0 6 105];
CFlowDates = [732678, NaN, 733408, 733774;
              732678, 733034, 733408, 734774];
```

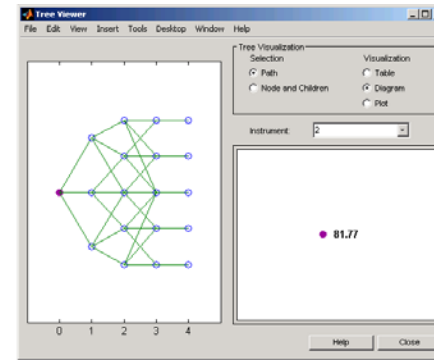
Use this information to compute the prices of the two cash flow instruments.

```
[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 `intenvset` for information on creating `RateSpec`.
`CFlowAmounts`	Number of instruments (`NINST`) by maximum number of cash flows (`MOSTCFS`) matrix with entries listing cash flow amounts corresponding to each date in `CFlowDates`. Each row is a list of cash flow values for one instrument. If an instrument has fewer than `MOSTCFS` cash flows, the end of the row is padded with `NaN`s.
`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. • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European)

- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

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

Purpose

Price European simple chooser options using Black-Scholes model

Syntax

```
Price = chooserbybls(RateSpec, StockSpec, Settle,
Maturity, Strike)
```

Arguments

`RateSpec`	The annualized continuously compounded rate term structure. For information on the interest rate specification, see `intenvset`.
`StockSpec`	Stock specification. See `stockspec`.
`Settle`	`NINST`-by-`1` vector of settlement or trade dates.
`Maturity`	`NINST`-by-`1` vector of maturity dates.
`Strike`	`NINST`-by-`1` vector of strike price values.
`ChooseDate`	`NINST`-by-`1` vector of chooser dates.

Description

`Price = chooserbybls(RateSpec, StockSpec, Settle, Maturity, Strike)` computes the price for European simple chooser options using the Black-Scholes model.

`Price` is a `NINST`-by-`1` vector of expected prices.

Note Only dividends of type `continuous` can be considered for choosers.

Examples

Consider a European chooser option with an exercise price of $60 on June 1, 2007. The option expires on December 2, 2007. Assume the underlying stock provides a continuous dividend yield of 5% per annum, is trading at $50, and has a volatility of 20% per annum. The annualized continuously compounded risk-free rate is 10% per annum. Assume that the choice must be made on August 31, 2007. Using this data:

```
AssetPrice = 50;
Strike = 60;
Settlement = 'Jun-1-2007';
Maturity = 'Dec-2-2007';
ChooseDate = 'Aug-31-2007';
RiskFreeRate = 0.1;
Sigma = 0.20;
Yield = 0.05
```

Define the `RateSpec` and `StockSpec`:

```
RateSpec = intenvset('Compounding', -1, 'Rates', RiskFreeRate, 'StartDates',...
Settlement, 'EndDates', Maturity);
StockSpec = stockspec(Sigma, AssetPrice,'continuous',Yield);
```

Price the chooser option:

```
Price  = chooserbybls(RateSpec, StockSpec, Settlement, Maturity,...
Strike, ChooseDate)

Price =

8.9308
```

References

Rubinstein, Mark, "Options for the Undecided," *Risk 4*, 1991.

See Also

`blsprice`, `intenvset`

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

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`

compoundbyeqp

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`

Purpose

Price compound options using 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(ITTTree, 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`

crrprice

Purpose

Instrument prices from CRR tree

Syntax

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

Arguments

`CRRTree`	Interest-rate tree structure created by `crrtree`.
`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. For more information about how to create the `InstSet` structure, see `instadd`.
`Options`	(Optional) Derivatives pricing options structure created with `derivset`.

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

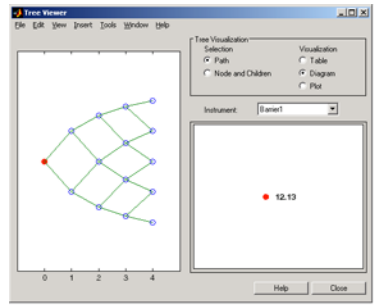
Load the CRR tree and instruments from the data file `deriv.mat`. Price the barrier and lookback options contained in the instrument set.

```
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'
```

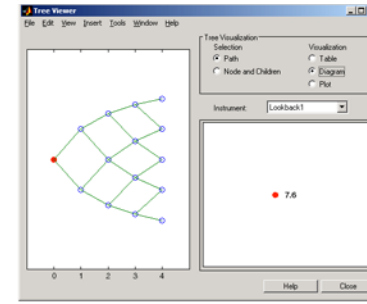
```
PTree: {1x5 cell}
 tObs: [0 1 2 3 4]
 dObs: [731582 731947 732313 732678 733043]
```

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

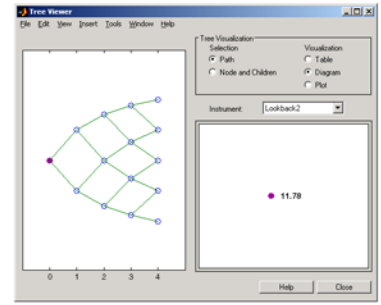
```
treeviewer(PriceTree, CRRSubSet)
```



Barrier1



Lookback1



Lookback2

See Also

`crrsens`, `crrtree`, `instadd`

Purpose

Instrument prices and sensitivities from CRR tree

Syntax

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

Arguments

`CRRTree`	Interest-rate tree structure created by `crrtree`.
`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] = crrsens(CRRTree, 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
```

See Also

crrprice, crrtree

crrtimespec

Purpose

Specify time structure for CRR tree

Syntax

```
TimeSpec = crrtimespec(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 = 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 1 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]
```

See Also

crrtree, stockspec

Purpose

Construct CRR stock tree

Syntax

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

Arguments

`StockSpec`	Stock specification. See `stockspec` 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 CRR binomial tree. See `crrtimespec` 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

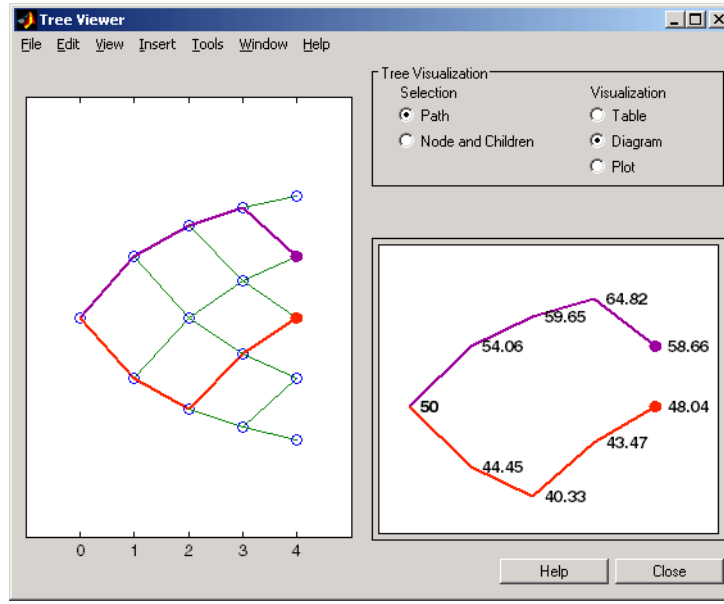
Using the data provided, create a stock specification (`StockSpec`), rate specification (`RateSpec`), and tree time layout specification (`TimeSpec`). Then use these specifications to create a CRR tree with `crrtree`.

```
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
               tObs: [0 0.2493 0.4986 0.7479 0.9972]
               dObs: [731582 731673 731764 731855 731946]

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 `treeviewer` 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

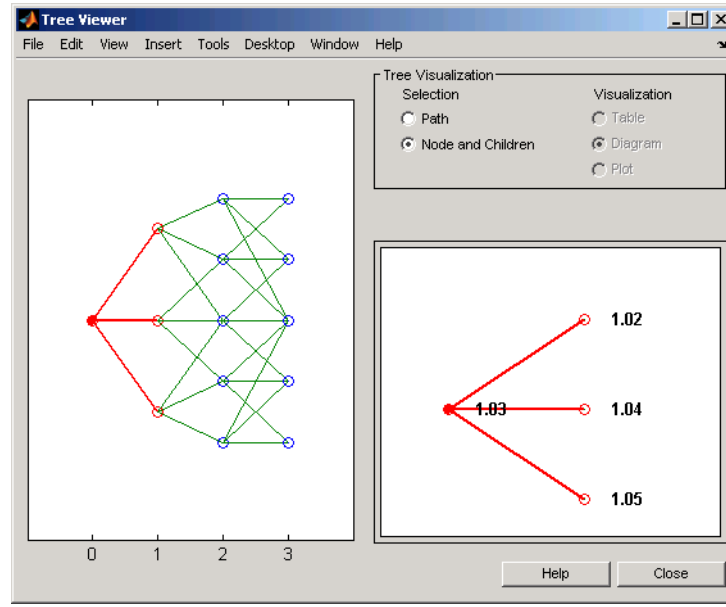
Convert a Hull-White tree using inverse-discount notation to a Hull-White tree displaying interest-rate notation.

```
load deriv.mat;

HWTree

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}

HWTree.FwdTree{1}
```

```
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)
```

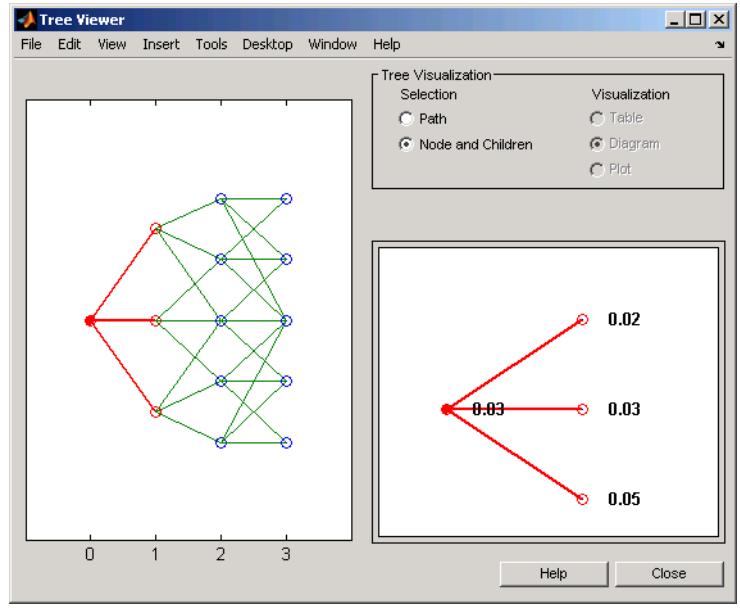


Use `cvtree` to convert the inverse-discount notation to interest-rate notation.

```
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 `treeviewer` to display the converted tree, showing the path of interest rates expressed as forward rates.



See Also

`disc2rate`, `rate2disc`

date2time

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`	(Optional) Scalar value representing the rate at which the input zero rates were compounded when annualized. This argument determines the formula for the discount factors:
	`Compounding` = `1`, `2`, `3`, `4`, `6`, `12` (Default = `2`.)
	`Disc = (1 + Z/F)^(-T)`, where `F` is the compounding frequency, `Z` is the zero rate, and `T` is the time in periodic units; for example, `T = F` is 1 year.
	`Compounding` = `365`
	`Disc = (1 + Z/F)^(-T)`, where `F` is the number of days in the basis year and `T` is a number of days elapsed computed by basis.
	`Compounding` = `-1`
	`Disc = exp(-T*Z)`, where `T` is time in years.

`Basis` (Optional) Day-count basis of the instrument. A vector of integers.

- 0 = actual/actual (default)
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

`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

`[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`

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
```

Remarks

This function is identical to the `datedisp` function in Financial Toolbox software.

See Also

`datenum`, `datestr` in Financial Toolbox documentation

Purpose

Get derivatives pricing options

Syntax

```
Value = derivget(Options, 'Parameter')
```

Arguments

`Options`	Existing options specification structure, probably created from previous call to `derivset`.
`Parameter`	Must be `'Diagnostics'`, `'Warnings'`, `'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 =

unenhanced
```

See Also

`barrierbycrr`, `barrierbyeqp`, `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 `derivset`.
`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'`: • `'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.

```
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`

disc2rate

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 1 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`. `StartDates` must be earlier than `EndDates`.
`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. • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ICMA) • 9 = actual/360 (ICMA) • 10 = actual/365 (ICMA)

- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

`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, `Tsemi`, by the formula `T = Tsemi/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 interpretation; omitting `ValuationDate` invokes the default time interpretations.

See Also

`rate2disc`, `ratetimes`

eqpprice

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

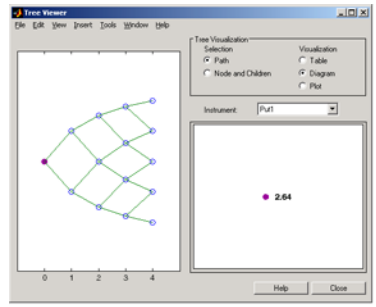
Load the EQP tree and instruments from the data file `deriv.mat`. Price the put options contained in the instrument set.

```
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}
```

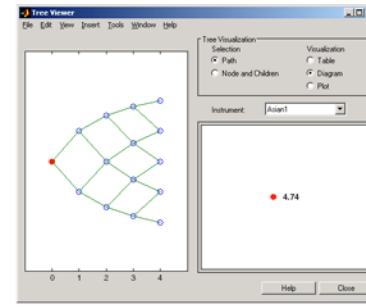
```
tObs: [0 1 2 3 4]
dObs: [731582 731947 732313 732678 733043]
```

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

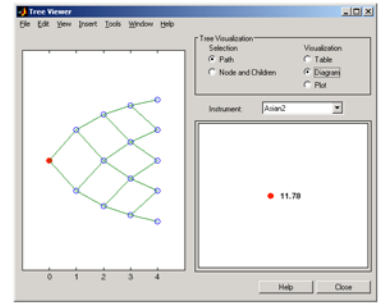
```
treeviewer(PriceTree, EQPSubSet)
```



Put1



Asian11



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'`, and `'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
```

See Also

eqpprice, eqptree

eqptimespec

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 1 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]
```

See Also

eqptree, stockspec

Purpose

Construct EQP stock tree

Syntax

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

Arguments

`StockSpec`	Stock specification. See `stockspec` 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

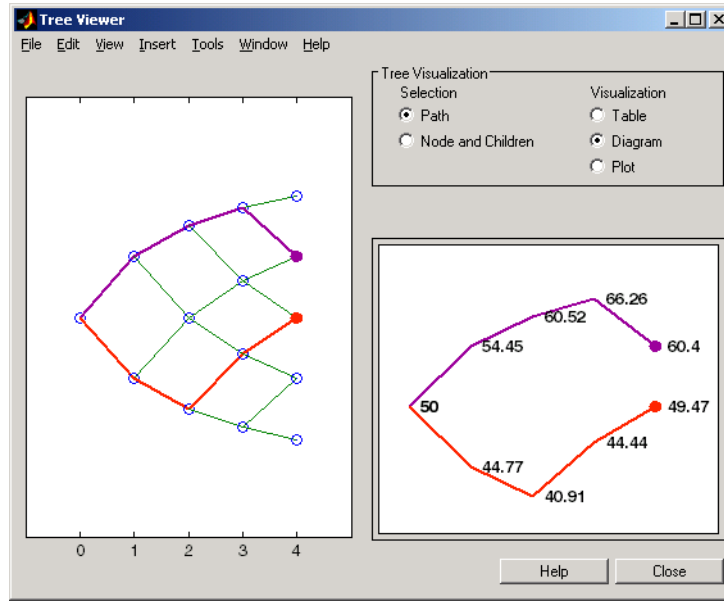
Using the data provided, create a stock specification (`StockSpec`), rate specification (`RateSpec`), and tree time layout specification (`TimeSpec`). Then use these specifications to create a CRR tree with `crrtree`.

```
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
             tObs: [0 0.2493 0.4986 0.7479 0.9972]
             dObs: [731582 731673 731764 731855 731946]

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(BDTTree, CouponRate, Settle,
Maturity, Reset, Basis, Principal, Options, EndMonthRule)
```

Arguments

`BDTTree`	Interest-rate tree structure created by `bdttree`.
`CouponRate`	Decimal annual rate.
`Settle`	Settlement dates. Number of instruments (`NINST`)-by-`1` vector of dates representing the settlement dates of the fixed-rate note.
`Maturity`	`NINST`-by-`1` vector of dates representing the maturity dates of the fixed-rate note.
`Reset`	(Optional) `NINST`-by-`1` vector representing the frequency of payments per year. Default = `1`.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers.

- 0 = actual/actual (default)
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)

- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

`Principal`	(Optional) The notional principal amount. Default = `100`.
`Options`	(Optional) Derivatives pricing options structure created with `derivset`.
`EndMonthRule`	(Optional) `NINST`-by-`1` vector representing the End-of-month rule. Default = `1`.

Description

`[Price, PriceTree] = fixedbybdt(BDTTree, CouponRate, Settle, Maturity, Reset, Basis, Principal, Options, EndMonthRule)` 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 `BDTTree`. The `BDTTree` 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(BDTTree, 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, EndMonthRule)
```

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.

- 0 = actual/actual (default)
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)

- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

`Principal`	(Optional) `NINST`-by-`1` vector of the notional principal amount. Default = `100`.
`Options`	(Optional) Derivatives pricing options structure created with `derivset`.
`EndMonthRule`	(Optional) `NINST`-by-`1` vector representing the End-of-month rule. Default = `1`.

Description

`[Price, PriceTree] = fixedbybk(BKTree, CouponRate, Settle, Maturity, Reset, Basis, Principal, Options, EndMonthRule)` 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';
```

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`

fixedbyhjm

Purpose

Price fixed-rate note from HJM interest-rate tree

Syntax

```
[Price, PriceTree] = fixedbyhjm(HJMTree, CouponRate, Settle,
Maturity, Reset, Basis, Principal, Options, EndMonthRule)
```

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.

- 0 = actual/actual (default)
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)

- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

`Principal`	(Optional) The notional principal amount. Default = `100`.
`Options`	(Optional) Derivatives pricing options structure created with `derivset`.
`EndMonthRule`	(Optional) `NINST`-by-`1` vector representing the End-of-month rule. Default = `1`.

Description

`[Price, PriceTree] = fixedbyhjm(HJMTree, CouponRate, Settle, Maturity, Reset, Basis, Principal, Options, EndMonthRule)` 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] = fixedbyhw(HWTree, CouponRate, Settle,
Maturity, Reset, Basis, Principal, Options, EndMonthRule)
```

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.

- 0 = actual/actual (default)
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)

- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

`Principal`	(Optional) `NINST`-by-`1` vector of the notional principal amount. Default = `100`.
`Options`	(Optional) Derivatives pricing options structure created with `derivset`.
`EndMonthRule`	(Optional) `NINST`-by-`1` vector representing the End-of-month rule. Default = `1`.

Description

`[Price, PriceTree] = fixedbyhw(HWTree, CouponRate, Settle, Maturity, Reset, Basis, Principal, Options, EndMonthRule)` 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';
```

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, EndMonthRule)
```

Arguments

`RateSpec`	Structure containing the properties of an interest-rate structure. See `intenvset` for information on creating `RateSpec`.
`CouponRate`	Decimal annual rate.
`Settle`	Settlement date. `Settle` must be earlier than `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.

- 0 = actual/actual (default)
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)

- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

`Principal`	(Optional) The notional principal amount. Default = `100`.
`EndMonthRule`	(Optional) `NINST`-by-`1` vector representing the End-of-month rule. Default = `1`.

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, EndMonthRule)` 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`

Purpose

Price floating-rate note from BDT interest-rate tree

Syntax

```
[Price, PriceTree] = floatbybdt(BDTTree, Spread,
Settle, Maturity, Reset, Basis, Principal, Options,
EndMonthRule)
```

Arguments

`BDTTree`	Interest-rate tree structure created by `bdttree`.
`Spread`	Number of instruments (`NINST`)-by-`1` vector of number of basis points over the reference rate.
`Settle`	Settlement dates. `NINST`-by-`1` vector of dates representing the settlement dates of the floating-rate note.
`Maturity`	`NINST`-by-`1` vector of dates representing the maturity dates of the floating-rate note.
`Reset`	(Optional) `NINST`-by-`1` vector representing the frequency of payments per year. Default = `1`.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ICMA)

- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

`Principal`	(Optional) `NINST`-by-`1` vector of the notional principal amount. Default = `100`.
`Options`	(Optional) Derivatives pricing options structure created with `derivset`.
`EndMonthRule`	(Optional) `NINST`-by-`1` vector representing the End-of-month rule. Default = `1`.

Description

`[Price, PriceTree] = floatbybdt(BDTTree, Spread, Settle, Maturity, Reset, Basis, Principal, Options,EndMonthRule)` computes the price of a floating-rate note from a BDT tree.

`Price` is an `NINST`-by-`1` vector of expected prices of the floating-rate note at time 0.

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

`PriceTree.PTree` contains the clean prices.

`PriceTree.AITree` contains the accrued interest.

`PriceTree.tObs` contains the observation times.

The `Settle` date for every floating-rate note is set to the `ValuationDate` of the BDT tree. The floating-rate note argument `Settle` is ignored.

Examples

Price a 20 basis point floating-rate note using a BDT interest-rate tree.

Load the file `deriv.mat`, which provides `BDTTree`. The `BDTTree` 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';
```

Use `floatbybdt` to compute the price of the note.

```
Price = floatbybdt(BDTTree, Spread, Settle, Maturity)

Price =

  100.4865
```

See Also

`bdttree`, `bondbybdt`, `capbybdt`, `cfbybdt`, `fixedbybdt`, `floorbybdt`, `swapbybdt`

floatbybk

Purpose

Price floating-rate note from Black-Karasinski interest-rate tree

Syntax

```
[Price, PriceTree] = floatbybk(BKTree, Spread,
Settle, Maturity, Reset, Basis, Principal, Options,
EndMonthRule)
```

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.

- 0 = actual/actual (default)
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)

- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

`Principal`	(Optional) `NINST`-by-`1` vector of the notional principal amount. Default = `100`.
`Options`	(Optional) Derivatives pricing options structure created with `derivset`.
`EndMonthRule`	(Optional) `NINST`-by-`1` vector representing the End-of-month rule. Default = `1`.

Description

`[Price, PriceTree] = floatbybk(BKTree, Spread, Settle, Maturity, Reset, Basis, Principal, Options,EndMonthRule)` 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`

Purpose

Price floating-rate note from HJM interest-rate tree

Syntax

```
[Price, PriceTree] = floatbyhjm(HJMTree, Spread, Settle,
Maturity, Reset, Basis, Principal, Options, EndMonthRule)
```

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.

- 0 = actual/actual (default)
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)

- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

`Principal`	(Optional) `NINST`-by-`1` vector of the notional principal amount. Default = `100`.
`Options`	(Optional) Derivatives pricing options structure created with `derivset`.
`EndMonthRule`	(Optional) `NINST`-by-`1` vector representing the End-of-month rule. Default = `1`.

Description

`[Price, PriceTree] = floatbyhjm(HJMTree, Spread, Settle, Maturity, Reset, Basis, Principal, Options, EndMonthRule)` 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';
```

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, EndMonthRule)
```

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. • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ICMA) • 9 = actual/360 (ICMA)

- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

`Principal`	(Optional) `NINST`-by-`1` vector of the notional principal amount. Default = `100`.
`Options`	(Optional) Derivatives pricing options structure created with `derivset`.
`EndMonthRule`	(Optional) `NINST`-by-`1` vector representing the End-of-month rule. Default = `1`.

Description

`[Price, PriceTree] = floatbyhw(HWTree, Spread, Settle, Maturity, Reset, Basis, Principal, Options, EndMonthRule)` 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`

Purpose

Price floating-rate note from set of zero curves

Syntax

```
Price = floatbyzero(RateSpec, Spread, Settle, Maturity, Reset,
Basis, Principal, EndMonthRule)
```

Arguments

`RateSpec`	Structure containing the properties of an interest-rate structure. See `intenvset` for information on creating `RateSpec`.
`Spread`	Number of basis points over the reference rate.
`Settle`	Settlement date. `Settle` must be earlier than `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.

- 0 = actual/actual (default)
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)

- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

`Principal`	(Optional) The notional principal amount. Default = `100`.
`EndMonthRule`	(Optional) `NINST`-by-`1` vector representing the End-of-month rule. Default = `1`.

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, EndMonthRule)` 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
```

See Also

`bondbyzero`, `cfbyzero`, `fixedbyzero`, `swapbyzero`

floorbybdt

Purpose

Price floor instrument from BDT interest-rate tree

Syntax

```
[Price, PriceTree] = floorbybdt(BDTTree, Strike, Settle, Maturity,
Reset, Basis, Principal, Options)
```

Arguments

`BDTTree`	Interest-rate tree structure created by `bdttree`.
`Strike`	Number of instruments (`NINST`)-by-`1` vector of rates at which the floor is exercised.
`Settle`	Settlement date. `NINST`-by-`1` vector of dates representing the settlement dates of the floor. The `Settle` date for every floor is set to the `ValuationDate` of the BDT tree. The floor argument `Settle` is ignored.
`Maturity`	`NINST`-by-`1` vector of dates representing the maturity dates of the floor.
`Reset`	(Optional) `NINST`-by-`1` vector representing the frequency of payments per year. Default = `1`.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese)

- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

`Principal`	(Optional) The notional principal amount. Default = `100`.
`Options`	(Optional) Derivatives pricing options structure created with `derivset`.

Description

`[Price, PriceTree] = floorbybdt(BDTTree, Strike, Settlement, Maturity, Reset, Basis, Principal, Options)` computes the price of a floor instrument from a BDT interest-rate tree.

`Price` is an `NINST`-by-`1` vector of the expected prices of the floor at time 0.

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

Examples

Example 1. Price a 10% floor instrument using a BDT interest-rate tree.

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

```
load deriv.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(BDTTree, Strike, Settle, Maturity)

Price =

   0.1770
```

Example 2. Here is a second example, showing the pricing of a 10% floor instrument using a newly created BDT tree.

First set the required arguments for the three needed specifications.

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

Next create the specifications.

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

Now create the BDT tree from the specifications.

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

Set the floor arguments. 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.

```
Price= floorbybdt(BDTTree, FloorStrike, Settlement, Maturity,...
FloorReset)

Price =

    0.0431
```

See Also

`bdttree`, `capbybdt`, `cfbybdt`, `swapbybdt`

floorbybk

Purpose

Price floor instrument from Black-Karasinski interest-rate tree

Syntax

```
[Price, PriceTree] = floorbybk(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 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. • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese)

- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

`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

Purpose

Price floors using Black option pricing model

Syntax

```
[FloorPrice, Floorlets] = floorbyblk(RateSpec, Strike, Settle,
Maturity, Volatility)
[FloorPrice, Floorlets] = floorbyblk(RateSpec, Strike, Settle,
Maturity, Volatility, 'Name1', Value1...)
```

Arguments

`RateSpec`	The annualized, continuously compounded rate term structure. For more information, see `intenvset`.
`Strike`	`NINST`-by-`1` vector of rates at which the floor is exercised, as a decimal number.
`Settle`	Scalar representing the settle date of the floor.
`Maturity`	Scalar representing the maturity date of the floor.
`Volatility`	`NINST`-by-`1` vector of volatilities.
`Reset`	(Optional) `NINST`-by-`1` vector representing the frequency of payments per year. Default is `1`.
`Principal`	(Optional) `NINST`-by-`1` vector representing the notional principal amount. Default is `100`.
`Basis`	`NINST`-by-`1` vector representing the basis used when annualizing the input forward rate. • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European)

- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

`ValuationDate` (Optional) Scalar representing the observation date of the investment horizons. The default is the `Settle` date.

Note All optional inputs are specified as matching parameter name/value pairs. The parameter name is specified as a character string, followed by the corresponding parameter value. You can specify parameter name/value pairs in any order. Names are case-insensitive and partial string matches are allowed provided no ambiguities exist.

Description

```
[FloorPrice, Floorlets] = floorbyblk(RateSpec, Strike,
Settle, Maturity, Volatility)
```

```
[FloorPrice, Floorlets] = floorbyblk(RateSpec, Strike,
Settle, Maturity, Volatility, 'Name1', Value1...)
```

Use `floorbyblk` to price floors using the Black option pricing model.

The outputs are:

- `FloorPrice` — `NINST`-by-`1` expected prices of the floor.
- `Floorlets` — `NINST`-by-`NCF` array of floorlets, padded with `NaN`s.

Examples

Consider an investor who gets into a contract that floors the interest rate on a $100,000 loan at 6% quarterly compounded for 3 months, starting on January 1, 2009'. Assuming that on January 1, 2008 the zero rate is 6.9394% continuously compounded and the volatility is 20%, use this data to compute the floor price.

Calculate the `RateSpec`:

```
ValuationDate = 'Jan-01-2008';
EndDates ='April-01-2010';
Rates = 0.069394;
Compounding = -1;
Basis = 1;

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

Compute the price of the cap:

```
Settle = 'Jan-01-2009'; % floor starts in a year
Maturity = 'April-01-2009';
Volatility = 0.20;
FloorRate = 0.06;
FloorReset = 4;
Principal=100000;

FloorPrice = floorbyblk(RateSpec, FloorRate, Settle, Maturity, Volatility,...
'Reset',FloorReset,'ValuationDate',ValuationDate,'Principal', Principal,...
'Basis', Basis)

FloorPrice =

   37.4864
```

See Also

`capbyblk`

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. • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese)

- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

`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)

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. • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese)

- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

`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`

gapbybls

Purpose

Calculate price of gap digital options using Black-Scholes model

Syntax

```
Price = gapbybls(RateSpec, StockSpec, Settle, Maturity,
OptSpec, Strike, StrikeThreshold)
```

Arguments

`RateSpec`	The annualized, continuously compounded rate term structure. For information on the interest rate specification, see `intenvset`.
`StockSpec`	Stock specification. See `stockspec`.
`Settle`	`NINST`-by-1 vector of settlement or trade dates.
`Maturity`	`NINST`-by-1 vector of maturity dates.
`OptSpec`	`NINST`-by-1 cell array of strings `'call'` or `'put'`.
`Strike`	`NINST`-by-1 vector of payoff strike price values.
`StrikeThreshold`	`NINST`-by-1 vector of strike values that determine if the option pays off.

Description

`Price = gapbybls(RateSpec, StockSpec, Settle, Maturity, OptSpec, Strike, StrikeThreshold)` computes gap option prices using the Black-Scholes option pricing model.

`Price` is a `NINST`-by-1 vector of expected option prices.

Examples

Consider a gap call and put options on a nondividend paying stock with a strike of 57 and expiring on January 1, 2008. On July 1, 2008 the stock is trading at 50. Using this data, compute the price of the option if the risk-free rate is 9%, the strike threshold is 50, and the volatility is 20%.

Create the `RateSpec`:

```
Settle = 'Jan-1-2008';
Maturity = 'Jul-1-2008';
```

```
Compounding = -1;
Rates = 0.09;
RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle,...
'EndDates', Maturity, 'Rates', Rates, 'Compounding', Compounding, 'Basis', 1);
```

Define the StockSpec:

```
AssetPrice = 50;
Sigma = .2;
StockSpec = stockspec(Sigma, AssetPrice);
```

Define the call and put options:

```
OptSpec = {'call'; 'put'};
Strike = 57;
StrikeThreshold = 50;
```

Calculate the price:

```
Pgap = gapbybls(RateSpec, StockSpec, Settle, Maturity, OptSpec,...
Strike, StrikeThreshold)

Pgap =

   -0.0053
    4.4866
```

See Also

assetbybls, cashbybls, gapsensbybls, supersharebybls

gapsensbybls

Purpose

Calculate price and sensitivities of gap digital options using Black-Scholes model

Syntax

```
PriceSens = gapsensbybls(RateSpec, StockSpec, Settle,
Maturity, OptSpec, Strike, StrikeThreshold)
PriceSens = gapsensbybls(RateSpec, StockSpec, Settle,
Maturity, OptSpec, Strike, StrikeThreshold, OutSpec)
```

Arguments

`RateSpec`	The annualized, continuously compounded rate term structure. For information on the interest rate specification, see `intenvset`.
`StockSpec`	Stock specification. See `stockspec`.
`Settle`	`NINST`-by-`1` vector of settlement or trade dates.
`Maturity`	`NINST`-by-`1` vector of maturity dates.
`OptSpec`	`NINST`-by-`1` cell array of strings `'call'` or `'put'`.
`Strike`	`NINST`-by-`1` vector of strike price values.
`StrikeThreshold`	`NINST`-by-`1` vector of strike values that determine if the option pays off.
`OutSpec`	(Optional) All optional inputs are specified as matching parameter name/value pairs. The parameter name is specified as a character string, followed by the corresponding parameter value. You can specify parameter name/value pairs may in any order. Names are case-insensitive and partial string matches are allowed provided no ambiguities exist. Valid parameter names are: • `NOUT`-by-`1` or `1`-by-`NOUT` cell array of strings indicating the nature and order of the outputs for the function. Possible values are `'Price'`,

'Delta', 'Gamma', 'Vega', 'Lambda', 'Rho', 'Theta', or 'All'.

For example, OutSpec = {'Price'; 'Lamba'; 'Rho'} specifies that the output should be Price, Lambda, and Rho, in that order.

To invoke from a function: [Price, Lambda, Rho] = gapsensbybls(..., 'OutSpec', {'Price', 'Lamba', 'Rho'})

OutSpec = {'All'} specifies that the output should be Delta, Gamma, Vega, Lambda, Rho, Theta, and Price, in that order. This is the same as specifying OutSpec as OutSpec = {'Delta', 'Gamma', 'Vega', 'Lambda', 'Rho', 'Theta', 'Price'};.

- Default is OutSpec = {'Price'}.

Description

PriceSens = gapsensbybls(RateSpec, StockSpec, Settle, Maturity, OptSpec, Strike, StrikeThreshold) computes gap option prices using the Black-Scholes option pricing model.

PriceSens = gapsensbybls(RateSpec, StockSpec, Settle, Maturity, OptSpec, Strike, StrikeThreshold, OutSpec) includes an OutSpec argument defined as parameter/value pairs, and computes gap option prices and sensitivities using the Black-Scholes option pricing model.

PriceSens is a NINST-by-1 vector of expected option prices and sensitivities.

Examples

Consider a gap call and put options on a nondividend paying stock with a strike of 57 and expiring on January 1, 2008. On July 1, 2008 the stock is trading at 50. Using this data, compute the price and sensitivity of the option if the risk-free rate is 9%, the strike threshold is 50, and the volatility is 20%.

Create the RateSpec:

```
Settle = 'Jan-1-2008';
Maturity = 'Jul-1-2008';
Compounding = -1;
Rates = 0.09;
RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle,...
'EndDates', Maturity, 'Rates', Rates, 'Compounding', Compounding, 'Basis', 1);
```

Define the StockSpec:

```
AssetPrice = 50;
Sigma = .2;
StockSpec = stockspec(Sigma, AssetPrice);
```

Define the call and put options:

```
OptSpec = {'call'; 'put'};
Strike = 57;
StrikeThreshold = 50;
```

Calculate the price:

```
Pgap = gapbybls(RateSpec, StockSpec, Settle, Maturity, OptSpec,...
Strike, StrikeThreshold)

Pgap =

   -0.0053
    4.4866
```

Compute the gamma and delta:

```
OutSpec = {'gamma'; 'delta'};
[Gamma ,Delta] = gapsensbybls(RateSpec, StockSpec, Settle, Maturity,...
OptSpec, Strike, StrikeThreshold, 'OutSpec', OutSpec)

Gamma =
```

```
    0.0724
    0.0724


Delta =

    0.2852
   -0.7148
```

See Also

gapbybls

hedgeopt

Purpose

Allocate optimal hedge for target costs or sensitivities

Syntax

```
[PortSens, PortCost, PortHolds] = hedgeopt(Sensitivities,
Price,  CurrentHolds, FixedInd, NumCosts, TargetCost,
TargetSens, ConSet)
```

Arguments

`Sensitivities`	Number of instruments (`NINST`) by number of sensitivities (`NSENS`) matrix of dollar sensitivities of each instrument. Each row represents a different instrument. Each column represents a different sensitivity.
`Price`	`NINST`-by-1 vector of portfolio instrument unit prices.
`CurrentHolds`	`NINST`-by-1 vector of contracts allocated to each instrument.
`FixedInd`	(Optional) Number of fixed instruments (`NFIXED`)-by-1 vector of indices of instruments to hold fixed. For example, to hold the first and third instruments of a 10 instrument portfolio unchanged, set `FixedInd = [1 3]`. Default = `[]`, no instruments held fixed.
`NumCosts`	(Optional) Number of points generated along the cost frontier when a vector of target costs (`TargetCost`) is not specified. The default is 10 equally spaced points between the point of minimum cost and the point of minimum exposure. When specifying `TargetCost`, enter `NumCosts` as an empty matrix `[]`.

`TargetCost`	(Optional) Vector of target cost values along the cost frontier. If `TargetCost` is empty, or not entered, `hedgeopt` evaluates `NumCosts` equally spaced target costs between the minimum cost and minimum exposure. When specified, the elements of `TargetCost` should be positive numbers that represent the maximum amount of money the owner is willing to spend to rebalance the portfolio.
`TargetSens`	(Optional) `1`-by-`NSENS` vector containing the target sensitivity values of the portfolio. When specifying `TargetSens`, enter `NumCosts` and `TargetCost` as empty matrices `[]`.
`ConSet`	(Optional) Number of constraints (`NCONS`) by number of instruments (`NINST`) matrix of additional conditions on the portfolio reallocations. An eligible `NINST`-by-`1` vector of contract holdings, `PortWts`, satisfies all the inequalities `A*PortWts <= b`, where `A = ConSet(:,1:end-1)` and `b = ConSet(:,end)`.

Notes

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 `NaN`s as placeholders.

Description

`[PortSens, PortCost, PortHolds] = hedgeopt(Sensitivities, Price, CurrentHolds, FixedInd, NumCosts, TargetCost, TargetSens, ConSet)` allocates an optimal hedge by one of two criteria:

- Minimize portfolio sensitivities (exposure) for a given set of target costs.
- Minimize the cost of hedging a portfolio given a set of target sensitivities.

Hedging involves the fundamental tradeoff between portfolio insurance and the cost of insurance coverage. This function 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 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 `FixedInd = 1`; the holdings in the first instrument are held fixed. If `NFIXED` instruments will not be changed, enter all their locations in the portfolio in a vector. If no instruments are to be held fixed, enter `FixedInd = []`.
`ConSet`	(Optional) Number of constraints (`NCONS`)-by-`NINST` matrix of additional conditions on the portfolio reallocations. An eligible `NINST`-by-`1` vector of contract holdings, `PortHolds`, satisfies all the inequalities `A*PortHolds <= b`, where `A = ConSet(:,1:end-1)` and `b = ConSet(:,end)`.

Description

`[PortSens, PortValue, PortHolds] = hedgeslf(Sensitivities, Price, CurrentHolds, FixedInd, ConSet)` allocates a self-financing hedge among a collection of instruments. hedgeslf finds the reallocation in a portfolio of financial instruments that hedges the portfolio against market moves and that is closest to being self-financing (maintaining constant portfolio value). By default the first instrument entered is hedged with the other instruments.

`PortSens` is a `1`-by-`NSENS` vector of portfolio dollar sensitivities. When a perfect hedge exists, `PortSens` is zeros. Otherwise, the best possible hedge is chosen.

`PortValue` is the total portfolio value (scalar). When a perfectly self-financing hedge exists, `PortValue` is equal to `dot(Price, CurrentWts)` of the initial portfolio.

`PortHolds` is an `NINST`-by-`1` vector of contracts allocated to each instrument. This is the reallocated portfolio.

Notes

1. The constraints `PortHolds(FixedInd) = CurrentHolds(FixedInd)` are appended to any constraints passed in `ConSet`. Pass `FixedInd = []` to specify all constraints through `ConSet`.

2. The default constraints generated by `portcons` are inappropriate, since they require the sum of all holdings to be positive and equal to one.

3. `hedgeself` first tries to find the allocations of the portfolio that make it closest to being self-financing, while reducing the sensitivities to 0. If no solution is found, it finds the allocations that minimize the sensitivities. If the resulting portfolio is self-financing, `PortValue` is equal to the value of the original portfolio.

Examples

Example 1. Perfect sensitivity cannot be reached.

```
Sens = [0.44  0.32; 1.0 0.0];
```

```
Price = [1.2; 1.0];
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

% WO(1) is not greater than 2.

[PortSens, PortValue, PortHolds] = hedgeslf(Sens, Price, WO,...
1, ConSet)

??? Error using ==> hedgeslf
Overly restrictive allocation constraints implied by ConSet and
by fixing the weight of instruments(s): 1
```

Example 3. Constraints are impossible to meet.

```
Sens = [0.44  0.32; 1.0 0.0];
Price = [1.2; 1.0];
WO = [1; 1];
ConSet = pcalims([2 2],[1 1]);

[PortSens, PortValue, PortHolds] = hedgeslf(Sens, Price, WO,...
[],ConSet)

??? Error using ==> hedgeslf
Overly restrictive allocation constraints specified in ConSet
```

See Also

`hedgeopt`

`lsqlin` in 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 `hjmtree` for information on creating `HJMTree`.
`InstSet`	Variable containing a collection of instruments. Instruments are categorized by type. Each type can have different data fields. The stored data field is a row vector or string for each instrument.
`Options`	(Optional) Derivatives pricing options structure created with `derivset`.

Description

`Price = hjmprice(HJMTree, InstSet, Options)` computes arbitrage-free prices for instruments using an interest-rate tree created with `hjmtree`. 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', 'OptEmBond', '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.
- optembndbyhjm: Price a bond with embedded option by an HJM tree.
- swapbyhjm: Price a swap from an HJM tree.
- swaptionbyhjm: Price a swaption from an HJM tree.

Examples

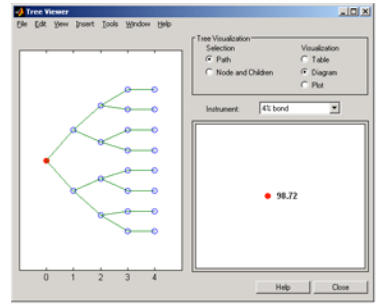
Load the HJM tree and instruments from the data file deriv.mat. Price the cap and bond instruments contained in the instrument set.

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

instdisp(HJMSubSet)

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

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

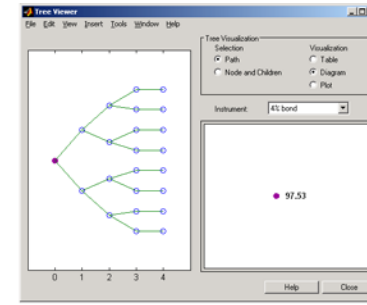
```
Warning: Not all cash flows are aligned with the tree. Result will
be approximated.

Price =

   98.7159
   97.5280
    6.2831

PriceTree =

    FinObj: 'HJMPriceTree'
     PBush: {[3x1 double]  [3x1x2 double]  [3x2x2 double]  [3x4x2 double]  [3x8 double]}
    AIBush: {[3x1 double]  [3x1x2 double]  [3x2x2 double]  [3x4x2 double]  [3x8 double]}
      tObs: [0 1 2 3 4]
```

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

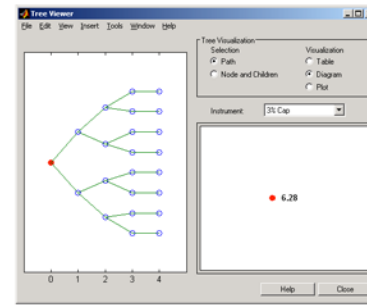
```
treeviewer(PriceTree, HJMSubSet)
```



First 4% Bond (Maturity 2003)



Second 4% Bond (Maturity 2004)



3% Cap

See Also

`hjmsens`, `hjmtree`, `hjmvolspec`, `instadd`, `intenvprice`, `intenvsens`

hjmsens

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 `hjmtree` for information on creating `HJMTree`.
`InstSet`	Variable containing a collection of instruments. Instruments are categorized by type. Each type can have different data fields. The stored data field is a row vector or string for each instrument.
`Options`	(Optional) Derivatives pricing options structure created with `derivset`.

Description

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

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

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

`Vega` is an `NINST`-by-`1` vector of vegas, representing the rate of change of instrument prices with respect to the changes in the volatility

$\sigma(t,T)$. Vega is computed by finite differences in calls to hjmtree. See hjmvolspec for information on the volatility process.

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

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

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

Examples

Load the tree and instruments from a data file. Compute Delta and Gamma for the cap and bond instruments contained in the instrument set.

```
load deriv.mat;
HJMSubSet = instselect(HJMInstSet,'Type', {'Bond', 'Cap'});
instdisp(HJMSubSet)

Index Type   CouponRate Settle      Maturity     Period  Name
...
1     Bond   0.04       01-Jan-2000 01-Jan-2003  1       4% bond
2     Bond   0.04       01-Jan-2000 01-Jan-2004  2       4% bond

Index Type Strike Settle      Maturity     CapReset...  Name ...
3     Cap  0.03   01-Jan-2000 01-Jan-2004  1            3% Cap

[Delta, Gamma] = hjmsens(HJMTree, HJMSubSet)

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

Delta =
```

```
   -272.6462
   -347.4315
    294.9700

  Gamma =

    1.0e+003 *

      1.0299
      1.6227
      6.8526
```

See Also

hjmprice, hjmtree, hjmvolspec, instadd

Purpose

Specify time structure for HJM interest-rate tree

Syntax

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

Arguments

`ValuationDate`	Scalar date marking the pricing date and first observation in the tree. Specify as serial date number or date string.
`Maturity`	Number of levels (depth) of the tree. A number of levels (`NLEVELS`)-by-`1` vector of dates marking the cash flow dates of the tree. Cash flows with these maturities fall on tree nodes. Maturity should be in increasing order.

`Compounding`	(Optional) Scalar value representing the rate at which the input zero rates were compounded when annualized. Default = `1`. This argument determines the formula for the discount factors:
	
	`Compounding` = `1`, `2`, `3`, `4`, `6`, `12`
	
	`Disc = (1 + Z/F)^(-T)`, where `F` is the compounding frequency, `Z` is the zero rate, and `T` is the time in periodic units; for example, `T = F` is 1 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.

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 `hjmvolspec` for information on the volatility process.
`RateSpec`	Interest-rate specification for the initial rate curve. See `intenvset` for information on declaring an interest-rate variable.
`TimeSpec`	Tree time layout specification. Defines the observation dates of the HJM tree and the compounding rule for date to time mapping and price-yield formulas. See `hjmtimespec` for information on the tree structure.

Description

`HJMTree = hjmtree(VolSpec, RateSpec, TimeSpec)` creates a structure containing time and forward-rate information on a bushy tree.

Examples

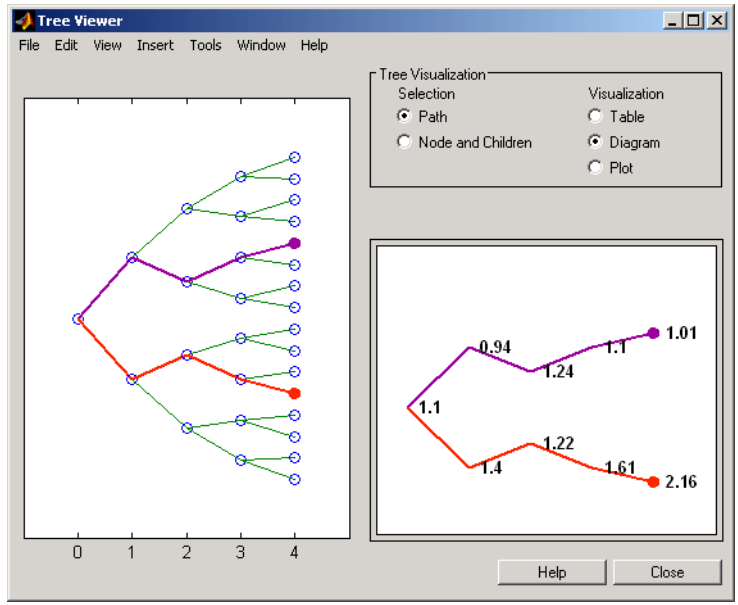
Using the data provided, create an HJM volatility specification (`VolSpec`), rate specification (`RateSpec`), and tree time layout specification (`TimeSpec`). Then use these specifications to create an HJM tree using `hjmtree`.

```
Compounding = 1;
ValuationDate = '01-01-2000';
StartDate = ['01-01-2000'; '01-01-2001'; '01-01-2002'; '01-01-2003'; '01-01-2004'];
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.

Argument	Description
CurveVol	Number of curves (NCURVES)-by1 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 an HJM forward-rate volatility process. Each factor is specified with one of the functional forms.

Volatility Specification	Formula
Constant	$\sigma(t,T)$ = Sigma_0
Stationary	$\sigma(t,T)$ = Vol(T-t) = Vol(Term)
Exponential	$\sigma(t,T)$ = Sigma_0*exp(-Lambda*(T-t))
Vasicek, Hull-White	$\sigma(t,T)$ = Sigma_0*exp(-Decay(T-t))
Proportional	$\sigma(t,T)$ = Prop(T-t)*max(SpotRate(t), MaxSpot)

The volatility process is $\sigma(t,T)$, where t is the observation time and T is the starting time of a forward rate. In a stationary process, the volatility term is T–t. Multiple factors can be specified sequentially.

The time values T, t, and `Term` are in coupon interval units specified by the `Compounding` input of `hjmtimespec`. For instance if `Compounding = 2`, `Term = 1` is a semiannual period (six months).

Examples

Example 1. Volatility is single-factor proportional.

```
CurveProp = [0.11765; 0.08825; 0.06865];
CurveTerm = [1; 2; 3];
VolSpec = hjmvolspec('Proportional', CurveProp, CurveTerm, 1e6)

 VolSpec =
          FinObj: 'HJMVolSpec'
    FactorModels: {'Proportional'}
      FactorArgs: {{1x3 cell}}
      SigmaShift: 0
      NumFactors: 1
       NumBranch: 2
         PBranch: [0.5000 0.5000]
     Fact2Branch: [-1 1]
```

Example 2. Volatility is two-factor exponential and constant.

```
VolSpec = hjmvolspec('Exponential', 0.1, 1, 'Constant', 0.2)

VolSpec =
          FinObj: 'HJMVolSpec'
    FactorModels: {'Exponential'  'Constant'}
      FactorArgs: {{1x2 cell}  {1x1 cell}}
      SigmaShift: 0
      NumFactors: 2
       NumBranch: 3
         PBranch: [0.2500 0.2500 0.5000]
```

```
                Fact2Branch: [2x3 double]
```

See Also

hjmtimespec, hjmtree

Purpose

Calibrate Hull-White tree using caps

Syntax

```
[Alpha, Sigma, OptimOut] = hwcalbycap(RateSpec, MarketStrike,
MarketMaturity, MarketVolatility, Strike, Settle, Maturity)
[Alpha, Sigma, OptimOut] = hwcalbycap(RateSpec, MarketStrike,
MarketMaturity, MarketVolatility, Strike, Settle, Maturity,
'Name1', Value1...)
```

Arguments

`RateSpec`	The annualized, continuously compounded rate term structure. For more information, see `intenvset`.
`MarketStrike`	`NSTRIKES`-by-`1` vector of market caplet strikes, as a decimal number.
`MarketMaturity`	`NMATS`-by-`1` vector of market caplet maturity dates.
`MarketVolatility`	`NSTRIKES`-by-`NMATS` matrix of market flat volatilities.
`Strike`	Scalar representing the rate at which the cap is exercised, as a decimal number.
`Settle`	Scalar representing the settle date of the cap.
`Maturity`	Scalar representing the maturity date of the cap.
`Reset`	(Optional) Scalar representing the frequency of payments per year. Default is `1`.
`Principal`	(Optional) Scalar representing the notional principal amount. Default is `100`.

`Basis`	`NINST`-by-`1` vector representing the basis used when annualizing the input forward rate. • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ICMA) • 9 = actual/360 (ICMA) • 10 = actual/365 (ICMA) • 11 = 30/360E (ICMA) • 12 = actual/actual (ISDA) • 13 = BUS/252
`ValuationDate`	(Optional) Scalar representing the observation date of the investment horizons. The default is the `Settle` date.
`LB`	(Optional) `2`-by-`1` vector of the lower bounds, defined as `[LBSigma; LBAlpha]`, used in the search algorithm function. Default is `LB =[0;0]`. For more information, see `lsqnonlin`.

`UB`	(Optional) 2-by-1 vector of the upper bounds, defined as `[UBSigma; LBAlpha]`, used in the search algorithm function. Default is `UB =[ ]`(unbound). For more information, see `lsqnonlin`.
`X0`	(Optional) 2-by-1 vector of the initial values, defined as `[Sigma0; Alpha0]`, used in the search algorithm function. Default is `X0 = [0.5;0.5]`. For more information, see `lsqnonlin`.
`OptimOptions`	(Optional) Structure with optimization parameters. For more information, see `optimset`.

Note All optional inputs are specified as matching parameter name/value pairs. The parameter name is specified as a character string, followed by the corresponding parameter value. You can specify parameter name/value pairs in any order. Names are case-insensitive and partial string matches are allowed provided no ambiguities exist.

Description

```
[Alpha, Sigma, OptimOut] = hwcalbycap(RateSpec,
MarketStrike,MarketMaturity, MarketVolatility, Strike,
Settle, Maturity)
```

```
[Alpha, Sigma, OptimOut] = hwcalbycap(RateSpec,
MarketStrike,MarketMaturity, MarketVolatility, Strike,
Settle, Maturity,'Name1', Value1...)
```

Use `hwcalbycap` to estimate the `Alpha` (mean reversion) and `Sigma` (volatility) using cap market data and the Hull-White model.

The outputs are:

- `Alpha` — Scalar representing the mean reversion value obtained from calibrating the cap using caplet market information.
- `Sigma` — Scalar representing the volatility value obtained from calibrating the cap using market caplet information.
- `OptimOut` — Structure with optimization results.

Examples

For an example, see "Calibrating the Hull-White Model Using Market Data" on page 2-42.

See Also

`capbyblk`, `hwcalbyfloor`, `hwtree`, `lsqnonlin`

Purpose

Calibrate Hull-White tree using floors

Syntax

```
[Alpha, Sigma, OptimOut] = hwcalbyfloor(RateSpec,
MarketStrike, MarketMaturity, MarketVolatility, Strike,
Settle, Maturity)
[Alpha, Sigma, OptimOut] = hwcalbyfloor(RateSpec,
MarketStrike, MarketMaturity, MarketVolatility, Strike,
Settle, Maturity, 'Name1', Value1...)
```

Arguments

`RateSpec`	The annualized, continuously compounded rate term structure. For more information, see `intenvset`.
`MarketStrike`	`NSTRIKES`-by-`1` vector of market floorlet strikes as a decimal number.
`MarketMaturity`	`NMATS`-by-`1` vector of market floorlet maturity dates.
`MarketVolatility`	`NSTRIKES`-by-`NMATS` matrix of market flat volatilities.
`Strike`	Scalar representing the rate at which the floor is exercised, as a decimal number.
`Settle`	Scalar representing the settle date of the floor.
`Maturity`	Scalar representing the maturity date of the floor.
`Reset`	(Optional) Scalar representing the frequency of payments per year. Default is `1`.

`Principal`	(Optional) Scalar representing the notional principal amount. Default is `100`.
`Basis`	`NINST`-by-`1` vector representing the basis used when annualizing the input forward rate. • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ICMA) • 9 = actual/360 (ICMA) • 10 = actual/365 (ICMA) • 11 = 30/360E (ICMA) • 12 = actual/actual (ISDA) • 13 = BUS/252
`ValuationDate`	(Optional) Scalar representing the observation date of the investment horizons. The default is the `Settle` date.

`LB`	(Optional) 2-by-1 vector of the lower bounds, defined as `[LBSigma; LBAlpha]`, used in the search algorithm function. Default is `LB =[0;0]`. For more information, see `lsqnonlin`.
`UB`	(Optional) 2-by-1 vector of the upper bounds, defined as `[UBSigma; LBAlpha]`, used in the search algorithm function. Default is `UB =[ ]`(unbound). For more information, see `lsqnonlin`.
`X0`	(Optional) 2-by-1 vector of the initial values, defined as `[Sigma0; Alpha0]`, used in the search algorithm function. Default is `X0 = [0.5;0.5]`. For more information, see `lsqnonlin`.
`OptimOptions`	(Optional) Structure with optimization parameters. For more information, see `optimset`.

Note All optional inputs are specified as matching parameter name/value pairs. The parameter name is specified as a character string, followed by the corresponding parameter value. You can specify parameter name/value pairs in any order. Names are case-insensitive and partial string matches are allowed provided no ambiguities exist.

Description

```
[Alpha, Sigma, OptimOut] =
hwcalbyfloor(RateSpec,MarketStrike, MarketMaturity,
MarketVolatility, Strike, Settle, Maturity)
```

```
[Alpha, Sigma, OptimOut] =
hwcalbyfloor(RateSpec,MarketStrike, MarketMaturity,
```

```
MarketVolatility, Strike, Settle, Maturity, 'Name1',
Value1...)
```

Use `hwcalbyfloor` to estimate the `Alpha` (mean reversion) and `Sigma` (volatility) using floor market data and the Hull-White model.

The outputs are:

- `Alpha` — Scalar representing the mean reversion value obtained from calibrating the floor using floorlet market information.
- `Sigma` — Scalar representing the volatility value obtained from calibrating the floor using floorlet market information.
- `OptimOut` — Structure with optimization results.

Examples

For an example, see "Calibrating the Hull-White Model Using Market Data" on page 2-42.

See Also

`floorbyblk`, `hwcalbycap`, `hwtree`, `lsqnonlin`

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'`, `'OptEmBond'`, `'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.
- `optembndbyhw`: Price a bond with embedded option by a Hull-White tree.
- `swapbyhw`: Price a swap from a Hull-White tree.
- `swaptionbyhw`: Price a swaption from a Hull-White tree.

Examples

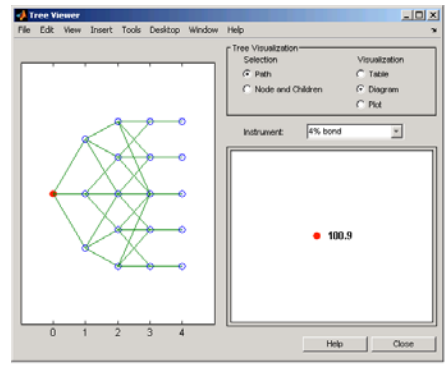
Load the HW tree and instruments from the data file `deriv.mat`. Price the cap and bond instruments contained in the instrument set.

```
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 =
```

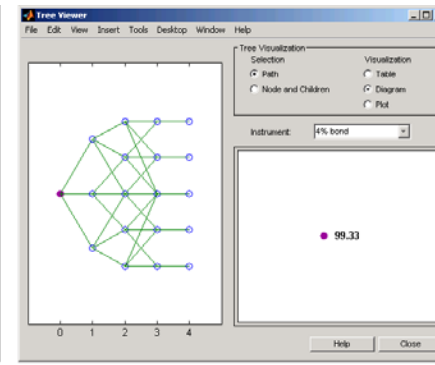
```
100.9188
 99.3296
  0.5837
```

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

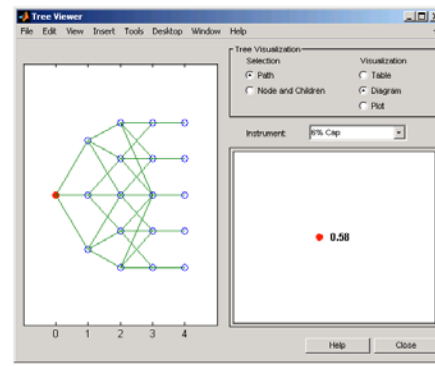
```
treeviewer(PriceTree, HWSubSet)
```



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'`, `'OptEmBond'`, `'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, hwvolspec, instadd

hwtimespec

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 1 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 = 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
     EndMonthRule: 1
```

See Also

`bktree`, `hwtree`, `hwvolspec`

hwtree

Purpose

Construct Hull-White interest-rate tree

Syntax

```
HWTree = hwtree(VolSpec, RateSpec, TimeSpec)
```

Arguments

`VolSpec`	Volatility process specification. See `hwvolspec` 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

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

Examples

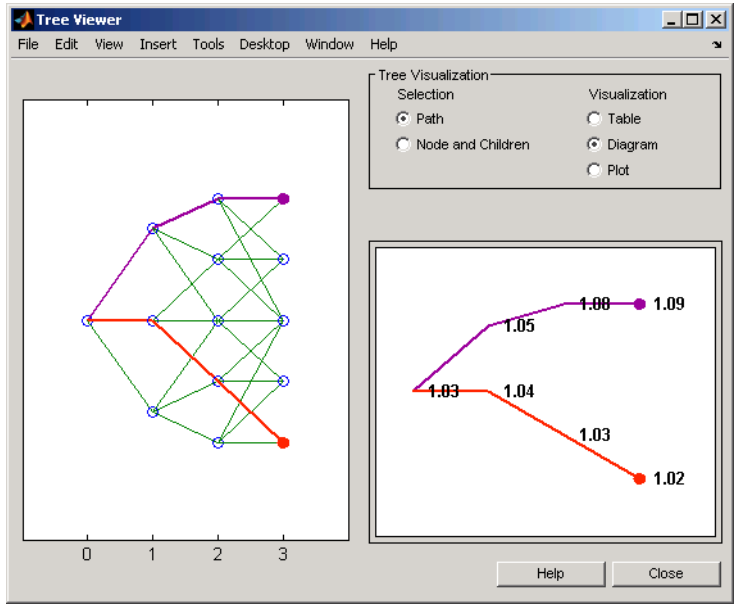
Using the data provided, create a Hull-White volatility specification (`VolSpec`), rate specification (`RateSpec`), and tree time layout specification (`TimeSpec`). Then use these specifications to create a Hull-White tree using `hwtree`.

```
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 = hwvolspec(ValuationDate, VolDates, VolCurve,...
AlphaDates, AlphaCurve);

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

HWTimeSpec = hwtimespec(ValuationDate, VolDates, Compounding);
HWTree = hwtree(HWVolSpec, RateSpec, HWTimeSpec)

HWTree =

      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(HWTree)
```



See Also

`hwcalbycap`, `hwcalbyfloor`, `hwprice`, `hwtimespec`, `hwvolspec`, `intenvset`

Purpose

Specify Hull-White interest-rate volatility process

Syntax

```
Volspec = hwvolspec(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 or scalar of yield volatility values in decimal form.
`AlphaDates`	`MPOINTS`-by-`1` vector of mean reversion end dates.
`AlphaCurve`	`MPOINTS`-by-`1` vector of positive mean reversion values or scalar in decimal form.
`InterpMethod`	(Optional) Interpolation method. Default is `'linear'`. See `interp1` for more information.

Note The number of points in `VolCurve` and `AlphaCurve` do not have to be the same.

Description

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

The volatility process is such that the variance of r(t + dt) - r(t) is defined as follows: V = (Volatility.^2 .* (1 - exp(-2*Alpha .* dt))) ./ (2 * Alpha). For more information on using Hull-White interest rate trees, see "Hull-White (HW) and Black-Karasinski (BK) Modeling" on page B-4.

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, hwcalbycap, hwcalbyfloor, interp1

Purpose

Calculate implied volatility using Bjerksund-Stensland 2002 option pricing model

Syntax

```
Volatility = impvbybjs(RateSpec, StockSpec, Settle,
Maturity, Strike, OptPrice, 'Name1', Value1...)
```

Arguments

`RateSpec`	The annualized continuously compounded rate term structure. For information on the interest rate specification, see `intenvset`.
`StockSpec`	Stock specification. See `stockspec`.
`Settle`	`NINST`-by-`1` vector of settlement or trade dates.
`Maturity`	`NINST`-by-`1` vector of maturity dates.
`OptSpec`	`NINST`-by-`1` cell array of strings `'call'` or `'put'`.
`Strike`	`NINST`-by-`1` vector of strike price values.
`OptPrice`	`NINST`-by-`1` vector of American option prices from which the implied volatility of the underlying asset are derived.

Note All optional inputs are specified as matching parameter name/parameter value pairs. The parameter name is specified as a character string, followed by the corresponding parameter value. You can specify parameter name/parameter value pairs in any order; names are case-insensitive and partial string matches are allowed provided no ambiguities exist.

`Limit`	(Optional) 1-by-2 positive vector representing the lower and upper bound of the implied volatility search interval. Default is `[0.1 10]`, or 10% to 1000% per annum.
`Tolerance`	(Optional) Positive scalar implied volatility termination tolerance. Default is `1e-6`.

Description

`Volatility = impvbybjs(RateSpec, StockSpec, Settle, Maturity, Strike, OptPrice, 'Name1', Value1...)` computes implied volatility using the Bjerksund-Stensland 2002 option pricing model.

`Volatility` is a `NINST`-by-1 vector of expected implied volatility values. If no solution is found, a `NaN` is returned.

Examples

Consider three American call options with exercise prices of $100 that expire on July 1, 2008. The underlying stock is trading at $100 on January 1, 2008 and pays a continuous dividend yield of 10%. The annualized continuously compounded risk-free rate is 10% per annum and the option prices are $4.063, $6.77 and $9.46. Using this data, calculate the implied volatility of the stock using the Bjerksund-Stensland 2002 option pricing model:

```
AssetPrice = 100;
Settle = 'Jan-1-2008';
Maturity = 'Jul-1-2008';
Strike = 100;
DivAmount = 0.1;
Rate = 0.1;
```

Define `RateSpec` and `StockSpec`:

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

```
StockSpec = stockspec(NaN, AssetPrice, {'continuous'}, DivAmount);
```

Calculate the implied volatility of the call options:

```
OptSpec = {'call'};
OptionPrice = [4.063;6.77;9.46];

ImpVol =  impvbybjs(RateSpec, StockSpec, Settle, Maturity, OptSpec,...
Strike, OptionPrice)

ImpvVol =

    0.1500
    0.2501
    0.3500
```

The implied volatility is 15% for the first call, and 25% and 35% for the second and third call options.

See Also

optstockbybjs, optstocksensbybjs

Purpose

Calculate implied volatility using Black option pricing model

Syntax

```
Volatility = impvbyblk(RateSpec, StockSpec, Settle,
Maturity, Strike, OptPrice, 'Name1', Value1...)
```

Arguments

`RateSpec`	The annualized continuously compounded rate term structure. For information on the interest rate specification, see `intenvset`.
`StockSpec`	Stock specification. See `stockspec`.
`Settle`	`NINST`-by-`1` vector of settlement or trade dates.
`Maturity`	`NINST`-by-`1` vector of maturity dates.
`OptSpec`	`NINST`-by-`1` cell array of strings `'call'` or `'put'`.
`Strike`	`NINST`-by-`1` vector of strike price values.
`OptPrice`	`NINST`-by-`1` vector of European option prices from which the implied volatility of the underlying asset are derived.

Note All optional inputs are specified as matching parameter name/parameter value pairs. The parameter name is specified as a character string, followed by the corresponding parameter value. You can specify parameter name/parameter value pairs in any order; names are case-insensitive and partial string matches are allowed provided no ambiguities exist.

`Limit`	(Optional) Positive scalar representing the upper bound of the implied volatility search interval. Default is `10`, or 1000% per annum.
`Tolerance`	(Optional) Positive scalar implied volatility termination tolerance. Default is `1e-6`.

Description

`Volatility = impvbyblk(RateSpec, StockSpec, Settle, Maturity, Strike, OptPrice, 'Name1', Value1...)` computes implied volatility using the Black option pricing model.

`Volatility` is a `NINST`-by-`1` vector of expected implied volatility values. If no solution is found, a `NaN` is returned.

Examples

Consider a European call and put options on a futures contract with exercise prices of $30 for the put and $40 for the call that expire on September 1, 2008. Assume that on May 1, 2008 the contract is trading at $35. The annualized continuously compounded risk-free rate is 5% per annum. What are the implied volatilities of the stock, if on that date, the call price is $1.14 and the put price is $0.82

```
AssetPrice = 35;
Strike = [30; 40];
Rates = 0.05;
Settle = 'May-01-08';
Maturity = 'Sep-01-08';
```

Create `RateSpec` and `StockSpec`:

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

StockSpec = stockspec(NaN, AssetPrice);
```

Define the options:

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

Calculate the implied volatility of the options:

```
Price = [1.14;0.82];
Volatility = impvbyblk(RateSpec, StockSpec, Settle, Maturity, OptSpec,...
Strike, Price)
```

```
Volatility =

    0.4052
    0.3021
```

The implied volatility would be 41% and 30%.

See Also

`optstockbyblk`, `optstocksensbyblk`

Purpose

Calculate implied volatility using Black-Scholes option pricing model

Syntax

```
Volatility = impvbybls(RateSpec, StockSpec, Settle,
Maturity, Strike, OptPrice, 'Name1', Value1...)
```

Arguments

`RateSpec`	The annualized continuously compounded rate term structure. For information on the interest rate specification, see `intenvset`.
`StockSpec`	Stock specification. See `stockspec`.
`Settle`	`NINST`-by-`1` vector of settlement or trade dates.
`Maturity`	`NINST`-by-`1` vector of maturity dates.
`OptSpec`	`NINST`-by-`1` cell array of strings `'call'` or `'put'`.
`Strike`	`NINST`-by-`1` vector of strike price values.
`OptPrice`	`NINST`-by-`1` vector of European option prices from which the implied volatility of the underlying asset are derived.

Note All optional inputs are specified as matching parameter name/parameter value pairs. The parameter name is specified as a character string, followed by the corresponding parameter value. You can specify parameter name/parameter value pairs in any order; names are case-insensitive and partial string matches are allowed provided no ambiguities exist.

`Limit`	(Optional) Positive scalar representing the upper bound of the implied volatility search interval. Default is `10`, or 1000% per annum.
`Tolerance`	(Optional) Positive scalar implied volatility termination tolerance. Default is `1e-6`.

Description

`Volatility = impvbybls(RateSpec, StockSpec, Settle, Maturity, Strike, OptPrice, 'Name1', Value1...)` computes implied volatility using the Black-Scholes option pricing model.

`Volatility` is a `NINST`-by-`1` vector of expected implied volatility values. If no solution is found, a `NaN` is returned.

Examples

Consider a European call and put options with an exercise price of $40 that expires on June 1, 2008. The underlying stock is trading at $45 on January 1, 2008 and the risk-free rate is 5% per annum. The option price is $7.10 for the call and $2.85 for the put. Using this data, calculate the implied volatility of the European call and put using the Black-Scholes option pricing model:

```
AssetPrice = 45;
Settlement = 'Jan-01-2008';
Maturity = 'June-01-2008';
Strike = 40;
Rates = 0.05;
OptionPrice = [7.10; 2.85];
OptSpec = {'call';'put'};
```

Define `RateSpec` and `StockSpec` :

```
RateSpec = intenvset('ValuationDate', Settlement, 'StartDates', Settlement,...
'EndDates', Maturity, 'Rates', Rates, 'Compounding', -1, 'Basis', 1);

StockSpec = stockspec(NaN, AssetPrice);
```

Calculate the implied volatility of the options:

```
ImpvVol =  impvbybls(RateSpec, StockSpec, Settlement, Maturity, OptSpec,...
Strike, OptionPrice)

ImpvVol =
```

```
    0.3175
    0.4878
```

The implied volatility is 31.75% for the call and 48.78% for the put.

See Also

`optstockbybls`, `optstocksensbybls`

impvbyrgw

Purpose

Calculate implied volatility using Roll-Geske-Whaley option pricing model for American call option

Syntax

```
Volatility = impvbyrgw(RateSpec, StockSpec, Settle,
Maturity, Strike, OptPrice, 'Name1', Value1...)
```

Arguments

`RateSpec`	The annualized continuously compounded rate term structure. For information on the interest rate specification, see `intenvset`.
`StockSpec`	Stock specification. See `stockspec`.
`Settle`	`NINST`-by-`1` vector of settlement or trade dates.
`Maturity`	`NINST`-by-`1` vector of maturity dates.
`Strike`	`NINST`-by-`1` vector of strike price values.
`OptPrice`	`NINST`-by-`1` vector of American call option prices from which the implied volatility of the underlying asset are derived.

Note All optional inputs are specified as matching parameter name/parameter value pairs. The parameter name is specified as a character string, followed by the corresponding parameter value. You can specify parameter name/parameter value pairs in any order; names are case-insensitive and partial string matches are allowed provided no ambiguities exist.

`Limit`	(Optional) Positive scalar representing the upper bound of the implied volatility search interval. Default is `10`, or 1000% per annum.
`Tolerance`	(Optional) Positive scalar implied volatility termination tolerance. Default is `1e-6`.

Description

`Volatility = impvbyrgw(RateSpec, StockSpec, Settle, Maturity, Strike, OptPrice, 'Name1', Value1...)` computes implied volatility using the Roll-Geske-Whaley option pricing model.

`Volatility` is a `NINST`-by-`1` vector of expected implied volatility values. If no solution is found, a `NaN` is returned.

Examples

Assume that on July 1, 2008 a stock is trading at $13 and pays a single cash dividend of $0.25 on November 1, 2008. The American call option with a strike price of $15 expires on July 1, 2009 and is trading at $1.346. The annualized continuously compounded risk-free rate is 5% per annum. Calculate the implied volatility of the stock using the Roll-Geske-Whaley option pricing model:

```
AssetPrice = 13;
Strike = 15;
Rates = 0.05;
Settle = 'July-01-08';
Maturity = 'July-01-09';
```

Define `StockSpec` and `RateSpec`:

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

StockSpec = stockspec(NaN, AssetPrice, {'cash'}, 0.25, {'Nov 1,2008'});
```

Calculate the implied volatility of the option:

```
Price = [1.346];
Volatility = impvbyrgw(RateSpec, StockSpec, Settle, Maturity, Strike, Price)

Volatility =

    0.3539
```

See Also

`optstockbyrgw`, `optstocksensbyrgw`

instadd

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 = instadd('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 with embedded option instrument. (See also `instoptembnd`.)

```
InstSet = instadd('OptEmBond', CouponRate, Settle, Maturity,
OptSpec, Strike, ExerciseDates, 'AmericanOpt', AmericanOpt, 'Period',
Period, 'Basis', Basis, 'EndMonthRule', EndMonthRule,'Face',Face,
'IssueDate', IssueDate, 'FirstCouponDate', FirstCouponDate,
'LastCouponDate', LastCouponDate, 'StartDate', StartDate)
```

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, EndMonthRule)
```

Floating-rate note instrument. (See also `instfloat`.)

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

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, EndMonthRule)
```

Swaption instrument. (See also `instswaption`.)

```
InstSet = instadd('Swaption', OptSpec, Strike, ExerciseDates,
Spread, Settle, Maturity, AmericanOpt, SwapReset, Basis, Principal)
```

To add instruments to an existing collection:

```
InstSet = instadd(InstSetOld, TypeString, Data1, Data2, ...)
```

Arguments

`InstSetOld`	Variable containing a collection of instruments. Instruments are classified by type; each type can have different data fields. The stored data field is a row vector or string for each instrument.

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.

instadd

Description

instadd stores instruments of types 'Asian', 'Barrier', 'Bond', 'Cap', 'CashFlow', 'Compound', 'Fixed', 'Float', 'Floor', 'Lookback', 'OptBond', 'OptStock', 'Swap', or 'Swaption'. This toolbox provides pricing and sensitivity routines for these instruments.

InstSet is an instrument set variable containing the new input data.

Examples

Create a portfolio with two cap instruments and a 4% bond.

```
Strike = [0.06; 0.07];
CouponRate = 0.04;
Settle = '06-Feb-2000';
Maturity = '15-Jan-2003';

InstSet = instadd('Cap', Strike, Settle, Maturity);
InstSet = instadd(InstSet, 'Bond', CouponRate, Settle, Maturity);
instdisp(InstSet)

Index Type Strike Settle      Maturity    CapReset Basis Principal
1     Cap  0.06   06-Feb-2000 15-Jan-2003 NaN      NaN   NaN
2     Cap  0.07   06-Feb-2000 15-Jan-2003 NaN      NaN   NaN

Index Type CouponRate Settle         Maturity ...
3     Bond 0.04       06-Feb-2000    15-Jan-2003...
```

See Also

instasian, instbarrier, instbond, instcap, instcf, instcompound, instfixed, instfloat, instfloor, instlookback, instoptbnd, instoptembnd, instoptstock, instswap, instswaption

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.

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 = instaddfield(InstSet,'Type','Futures',...
'FieldName',{'Delivery','F'},'FieldClass',{'date','dble'},...
'Data' ,{'01-Jul-99',104.4 });
instdisp(InstSet)
Index Type   Strike Price Opt
1     Option  95    12.2  Call
2     Option 100     9.2  Call
3     Option 105     6.8  Call

Index Type    Delivery       F
4     Futures 01-Jul-1999    104.4
```

Add a put option.

```
FN = instfields(InstSet,'Type','Option')
InstSet = instaddfield(InstSet,'Type','Option',...
'FieldName',FN, 'Data',{105, 7.4, 'Put'});
instdisp(InstSet)

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

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

Index Type   Strike Price Opt
5     Option 105     7.4  Put
```

Make a placeholder for another put.

```
InstSet = instaddfield(InstSet,'Type','Option',...
'FieldName','Opt','Data','Put')
instdisp(InstSet)
```

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

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

Index Type   Strike Price Opt
5     Option 105     7.4  Put
6     Option NaN     NaN  Put
```

Add a cash instrument.

```
InstSet = instaddfield(InstSet, 'Type', 'TBill',...
'FieldName','Price','Data',99)
instdisp(InstSet)

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

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

Index Type   Strike Price Opt
5     Option 105     7.4  Put
6     Option NaN     NaN  Put

Index Type  Price
7     TBill 99
```

See Also

instdisp, instget, instgetcell, instsetfield

Purpose

Construct Asian option

Syntax

```
InstSet = instasian(InstSet, OptSpec, Strike, Settle,
ExerciseDates, AmericanOpt, AvgType, AvgPrice, AvgDate)
[FieldList, ClassList, TypeString] = instasian
```

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.
`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.
`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. 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

Purpose

Construct barrier option

Syntax

```
InstSet = instbarrier(InstSet, OptSpec, Strike, Settle,
ExerciseDates, AmericanOpt, BarrierSpec, Barrier, Rebate)
[FieldList, ClassList, TypeString] = instbarrier
```

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.
`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.
`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) 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

`InstSet`	Instrument variable. This argument is specified only when adding bond instruments to an existing instrument set. See `instget` for more information on the `InstSet` variable.
`CouponRate`	Decimal number indicating the annual percentage rate used to determine the coupons payable on a bond.
`Settle`	Settlement date. A vector of serial date numbers or date strings. `Settle` must be earlier than `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.

- 0 = actual/actual (default)
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

`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`	(Future implementation)
`Face`	(Optional) Face or par value. Default = `100`.

Data arguments are number of instruments (`NINST`)-by-`1` vectors, scalar, or empty. Fill unspecified entries in vectors with `NaN`. Only one data argument is required to create the instrument. The others may be omitted or passed as empty matrices `[]`.

Description

`InstSet = instbond(InstSet, CouponRate, Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate, StartDate, Face)` creates a new instrument set containing bond instruments or adds bond instruments to a existing instrument set.

`[FieldList, ClassList, TypeString] = instbond` displays the classes.

`FieldList` is a number of fields (`NFIELDS`)-by-`1` cell array of strings listing the name of each data field for this instrument type.

`ClassList` is an `NFIELDS`-by-`1` cell array of strings listing the data class of each field. The class determines how arguments are parsed. Valid strings are `'dble'`, `'date'`, and `'char'`.

`TypeString` is a string specifying the type of instrument added. For a bond instrument, `TypeString = 'Bond'`.

See Also

`hjmprice`, `instaddfield`, `instdisp`, `instget`, `intenvprice`

instcap

Purpose

Construct cap instrument

Syntax

```
InstSet = instcap(InstSet, Strike, Settle, Maturity, Reset,
Basis, Principal)
[FieldList, ClassList, TypeString] = instcap
```

Arguments

`InstSet`	Instrument variable. This argument is specified only when adding cap instruments to an existing instrument set. See `instget` for more information on the `InstSet` variable.
`Strike`	Rate at which the cap is exercised, as a decimal number.
`Settle`	Settlement date. Serial date number representing the settlement date of the cap.
`Maturity`	Serial date number representing the maturity date of the cap.
`Reset`	(Optional) `NINST`-by-`1` vector representing the frequency of payments per year. Default = `1`.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese)

- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

`Principal` (Optional) The notional principal amount. Default = `100`.

Description

`InstSet = instcap(InstSet, Strike, Settle, Maturity, Reset, Basis, Principal)` creates a new instrument set containing cap instruments or adds cap instruments to an existing instrument set.

`[FieldList, ClassList, TypeString] = instcap` displays the classes.

`FieldList` is a number of fields (`NFIELDS`)-by-`1` cell array of strings listing the name of each data field for this instrument type.

`ClassList` is an `NFIELDS`-by-`1` cell array of strings listing the data class of each field. The class determines how arguments are parsed. Valid strings are `'dble'`, `'date'`, and `'char'`.

`TypeString` is a string specifying the type of instrument added. For a cap instrument, `TypeString = 'Cap'`.

See Also

`hjmprice`, `instaddfield`, `instbond`, `instdisp`, `instfloor`, `instswap`, `intenvprice`

instcf

Purpose

Construct cash flow instrument

Syntax

```
InstSet = instcf(InstSet, CFlowAmounts, CFlowDates, Settle,
Basis)
[FieldList, ClassList, TypeString] = instcf
```

Arguments

`InstSet`	Instrument variable. This argument is specified only when adding cash flow instruments to an existing instrument set. See `instget` for more information on the `InstSet` variable.
`CFlowAmounts`	Number of instruments (`NINST`) by maximum number of cash flows (`MOSTCFS`) matrix of cash flow amounts. Each row is a list of cash flow values for one instrument. If an instrument has fewer than `MOSTCFS` cash flows, the end of the row is padded with `NaN`s.
`CFlowDates`	`NINST`-by-`MOSTCFS` matrix of cash flow dates. Each entry contains the date of the corresponding cash flow in `CFlowAmounts`.
`Settle`	Settlement date on which the cash flows are priced.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA)

- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

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 metadata 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`

instcompound

Purpose

Construct compound option

Syntax

```
InstSet = instcompound(InstSet, UOptSpec, UStrike, USettle,
UExerciseDates, UAmericanOpt, COptSpec, CStrike, CSettle,
CExerciseDates, CAmericanOpt)
[FieldList, ClassList, TypeString] = instcompound
```

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.
`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

`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`

Purpose

Complement of instrument set by matching conditions

Syntax

```
ISubSet = instdelete(InstSet, 'FieldName', FieldList, 'Data',
DataList, 'Index', IndexSet, 'Type', TypeList)
```

Arguments

`InstSet`	Variable containing a collection of instruments. Instruments are classified by type; each type can have different data fields. The stored data field is a row vector or string for each instrument.
`FieldList`	String or number of fields (`NFIELDS`)-by-`1` cell array of strings listing the name of each data field to match with data values.
`DataList`	Number of values (`NVALUES`)-by-`M` array or `NFIELDS`-by-`1` cell array of acceptable data values for each field. Each row lists a data row value to search for in the corresponding `FieldList`. The number of columns is arbitrary and matching will ignore trailing `NaN`s or spaces.
`IndexSet`	(Optional) Number of instruments (`NINST`)-by-`1` vector restricting positions of instruments to check for matches. The default is all indices available in the instrument variable.
`TypeList`	(Optional) String or number of types (`NTYPES`)-by-`1` cell array of strings restricting instruments to match one of `TypeList` types. The default is all types in the instrument variable.

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.

```
ISet = instdelete(ExampleInst, 'Type','Option');
instdisp(ISet)

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

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

See Also

instaddfield, instfind, instget, instselect

instdisp

Purpose

Display instruments

Syntax

```
CharTable = instdisp(InstSet)
```

Arguments

`InstSet`	Variable containing a collection of instruments. See `instaddfield` for examples on constructing the variable.

Description

`CharTable = instdisp(InstSet)` creates a character array displaying the contents of an instrument collection, `InstSet`. If instdisp is called without output arguments, the table is displayed in the Command Window.

`CharTable` is a character array with a table of instruments in `InstSet`. For each instrument row, the `Index` and `Type` are printed along with the field contents. Field headers are printed at the tops of the columns.

Examples

Retrieve the instrument set `ExampleInst` from the data file `InstSetExamples.mat`. `ExampleInst` contains three types of instruments: `Option`, `Futures`, and `TBill`.

```
load InstSetExamples;
instdisp(ExampleInst)

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

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

Index Type   Strike Price Opt  Contracts
5     Option 105      7.4 Put  -1000
```

```
6     Option  95      2.9 Put      0

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

See Also

`datestr` in Financial Toolbox documentation

`num2str` in MATLAB Reference documentation

`instaddfield`, `instget`

instfields

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
```

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

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

Get the fields listed for type `'Option'`.

```
[FieldList, ClassList] = instfields(ExampleInst, 'Type',...
'Option')

FieldList =

    'Strike'
    'Price'
    'Opt'
    'Contracts'

ClassList =

    'dble'
    'dble'
    'char'
    'dble'
```

Get the fields listed for types `'Option'` and `'TBill'`.

```
FieldList = instfields(ExampleInst, 'Type', {'Option', 'TBill'})

FieldList =

    'Strike'
    'Opt'
    'Price'
    'Maturity'
    'Contracts'
```

Get all the fields listed in any type in the variable.

```
FieldList = instfields(ExampleInst)
FieldList =

    'Delivery'
    'F'
    'Strike'
    'Opt'
    'Price'
    'Maturity'
    'Contracts'
```

See Also

`instdisp`, `instlength`, `insttypes`

Purpose

Search instruments for matching conditions

Syntax

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

Arguments

`InstSet`	Variable containing a collection of instruments. Instruments are classified by type; each type can have different data fields. The stored data field is a row vector or string for each instrument.
`FieldList`	String or number of fields (`NFIELDS`)-by-`1` cell array of strings listing the name of each data field to match with data values.
`DataList`	Number of values (`NVALUES`)-by-`M` array or `NFIELDS`-by-`1` cell array of acceptable data values for each field. Each row lists a data row value to search for in the corresponding `FieldList`. The number of columns is arbitrary, and matching will ignore trailing `NaN`s or spaces.
`IndexSet`	(Optional) Number of instruments (`NINST`)-by-`1` vector restricting positions of instruments to check for matches. The default is all indices available in the instrument variable.
`TypeList`	(Optional) String or number of types (`NTYPES`)-by-`1` cell array of strings restricting instruments to match one of `TypeList` types. The default is all types in the instrument variable.

Argument value pairs can be entered in any order. The `InstSet` variable must be the first argument. `'FieldName'` and `'Data'` arguments must appear together or not at all.

Description

`IndexMatch = instfind(InstSet, 'FieldName', FieldList, 'Data', DataList,'Index', IndexSet, 'Type', TypeList)` returns indices of instruments matching `Type`, `Field`, or `Index` values.

`IndexMatch` is an `NINST`-by-`1` vector of positions of instruments matching the input criteria. Instruments are returned in `IndexMatch` if all the `Field`, `Index`, and `Type` conditions are met. An instrument meets an individual `Field` condition if the stored `FieldName` data matches any of the rows listed in the `DataList` for that `FieldName`.

Examples

Retrieve the instrument set `ExampleInst` from the data file `InstSetExamples.mat`. `ExampleInst` contains three types of instruments: `Option`, `Futures`, and `TBill`.

```
load InstSetExamples;
instdisp(ExampleInst)

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

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

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

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

Make a vector, `Opt95`, containing the indexes within `ExampleInst` of the options struck at 95.

```
Opt95 = instfind(ExampleInst, 'FieldName','Strike','Data','95')

Opt95 =
```

```
     1
     6
```

Locate the futures and Treasury bill instruments within `ExampleInst`.

```
Types = instfind(ExampleInst,'Type',{'Futures';'TBill'})

Types =

     4
     7
```

See Also

`instaddfield`, `instget`, `instgetcell`, `instselect`

instfixed

Purpose

Construct fixed-rate instrument

Syntax

```
InstSet = instfixed(InstSet, CouponRate, Settle, Maturity,
Reset, Basis, Principal, EndMonthRule)
[FieldList, ClassList, TypeString] = instfixed
```

Arguments

`InstSet`	Instrument variable. This argument is specified only when adding fixed-rate note instruments to an existing instrument set. See `instget` for more information on the `InstSet` variable.
`CouponRate`	Decimal annual rate.
`Settle`	Settlement date. Date string or serial date number representing the settlement date of the fixed-rate note.
`Maturity`	Date string or serial date number representing the maturity date of the fixed-rate note.
`Reset`	(Optional) `NINST`-by-`1` vector representing the frequency of payments per year. Default = `1`.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese)

- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

`Principal`	(Optional) The notional principal amount. Default = `100`.
`EndMonthRule`	(Optional) `NINST`-by-`1` vector representing the End-of-month rule. Default = `1`.

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, CouponRate, Settle, Maturity, Reset, Basis, Principal, EndMonthRule)` 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, EndMonthRule)
[FieldList, ClassList, TypeString] = instfloat
```

Arguments

`InstSet`	Instrument variable. This argument is specified only when adding floating-rate note instruments to an existing instrument set. See `instget` for more information on the `InstSet` variable.
`Spread`	Number of basis points over the reference rate.
`Settle`	Settlement date. Date string or serial date number representing the settlement date of the floating-rate note.
`Maturity`	Date string or serial date number representing the maturity date of the floating-rate note.
`Reset`	(Optional) `NINST`-by-`1` vector representing the frequency of payments per year. Default = `1`.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese)

- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

`Principal`	(Optional) The notional principal amount. Default = `100`.
`EndMonthRule`	(Optional) `NINST`-by-`1` vector representing the End-of-month rule. Default = `1`.

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, EndMonthRule)` creates a new instrument set containing floating-rate instruments or adds floating-rate instruments to an existing instrument set.

`[FieldList, ClassList, TypeString] = instfloat` displays the classes.

`FieldList` is a number of fields (`NFIELDS`)-by-`1` cell array of strings listing the name of each data field for this instrument type.

`ClassList` is an `NFIELDS`-by-`1` cell array of strings listing the data class of each field. The class determines how arguments are parsed. Valid strings are `'dble'`, `'date'`, and `'char'`.

`TypeString` is a string specifying the type of instrument added. For a floating-rate instrument, `TypeString = 'Float'`.

See Also

hjmprice, instaddfield, instbond, instcap, instdisp, instswap, intenvprice

instfloor

Purpose

Construct floor instrument

Syntax

```
InstSet = instfloor(InstSet, Strike, Settle, Maturity, Reset,
Basis, Principal)
[FieldList, ClassList, TypeString] = instfloor
```

Arguments

`InstSet`	Instrument variable. This argument is specified only when adding floor instruments to an existing instrument set. See `instget` for more information on the `InstSet` variable.
`Strike`	Rate at which the floor is exercised, as a decimal number.
`Settle`	Settlement date. A vector of serial date numbers or date strings. `Settle` must be earlier than `Maturity`.
`Maturity`	Maturity date. A vector of serial date numbers or date strings.
`Reset`	(Optional) `NINST`-by-`1` vector representing the frequency of payments per year. Default = `1`.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese)

- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

`Principal` (Optional) The notional principal amount. Default = `100`.

Description

`InstSet = instfloor(InstSet, Strike, Settle, Maturity, Reset, Basis, Principal)` creates a new instrument set containing floor instruments or adds floor instruments to an existing instrument set.

`[FieldList, ClassList, TypeString] = instfloor` displays the classes.

`FieldList` is a number of fields (`NFIELDS`)-by-`1` cell array of strings listing the name of each data field for this instrument type.

`ClassList` is an `NFIELDS`-by-`1` cell array of strings listing the data class of each field. The class determines how arguments are parsed. Valid strings are `'dble'`, `'date'`, and `'char'`.

`TypeString` is a string specifying the type of instrument added. For a floor instrument, `TypeString = 'Floor'`.

See Also

`hjmprice`, `instaddfield`, `instbond`, `instcap`, `instdisp`, `instswap`, `intenvprice`

instget

Purpose

Data from instrument variable

Syntax

```
[Data_1, Data_2,...,Data_n] = instget(InstSet, 'FieldName',
FieldList, 'Index', IndexSet, 'Type', TypeList)
```

Arguments

`InstSet`	Variable containing a collection of instruments. Instruments are classified by type; each type can have different data fields. The stored data field is a row vector or string for each instrument.
`FieldList`	(Optional) String or number of fields (`NFIELDS`)-by-`1` cell array of strings listing the name of each data field to match with data values. `FieldList` entries can also be either `'Type'` or `'Index'`; these return type strings and index numbers respectively. The default is all fields available for the returned set of instruments.
`IndexSet`	(Optional) Number of instruments (`NINST`)-by-`1` vector of positions of instruments to work on. If `TypeList` is also entered, instruments referenced must be one of `TypeList` types and contained in `IndexSet`. The default is all indices available in the instrument variable.
`TypeList`	(Optional) String or number of types (`NTYPES`)-by-`1` cell array of strings restricting instruments to match one of `TypeList` types. The default is all types in the instrument variable.

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,O] = instget(ExampleInst, 'Index', Ind, 'FieldName',...
{'Type', 'Opt'})

T =
```

```
Option
Futures
Option
TBill

O =

Call

Put
```

Create a string report of holdings `Type`, `Opt`, and `Value`.

```
rstring = [T, O, num2str(V(Ind))]

rstring =

Option Call   6800
Futures        NaN
Option Put   -7400
TBill          594
```

See Also

`instaddfield`, `instdisp`, `instgetcell`

Purpose

Data and context from instrument variable

Syntax

```
[DataList, FieldList, ClassList] =
instgetcell(InstSet,  'FieldName', FieldList, 'Index',
IndexSet, 'Type', TypeList)
```

Arguments

`InstSet`	Variable containing a collection of instruments. Instruments are classified by type; each type can have different data fields. The stored data field is a row vector or string for each instrument.
`FieldList`	(Optional) String or number of fields (`NFIELDS`)-by-`1` cell array of strings listing the name of each data field to match with data values. `FieldList` should not be either `Type` or `Index`; these field names are reserved. The default is all fields available for the returned set of instruments.
`IndexSet`	(Optional) Number of instruments (`NINST`)-by-`1` vector of positions of instruments to work on. If `TypeList` is also entered, instruments referenced must be one of `TypeList` types and contained in `IndexSet`. The default is all indices available in the instrument variable.
`TypeList`	(Optional) String or number of types (`NTYPES`)-by-`1` cell array of strings restricting instruments to match one of `TypeList` types. The default is all types in the instrument variable.

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
```

```
    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

instlength

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`

Purpose

Construct lookback option

Syntax

```
InstSet = instlookback(InstSet, OptSpec, Strike, Settle,
ExerciseDates, AmericanOpt)
[FieldList, ClassList, TypeString] = instlookback
```

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.
`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.
`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 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`

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`	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 `NaN`s. For an American option: `NINST`-by-`1` vector of strike price values for each option.
`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

`InstSet = instoptbnd(InstSet, BondIndex, OptSpec, Strike, ExerciseDates)` specifies a European or Bermuda option.

`InstSet = instoptbnd(InstSet, BondIndex, OptSpec, Strike, ExerciseDates, AmericanOpt)` specifies an American option if `AmericanOpt` is set to `1`. If `AmericanOpt` is not set to `1`, the function specifies a European or Bermuda option.

`[FieldList, 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`

instoptembnd

Purpose

Constructor for 'Type', 'OptEmBond' bond with embedded option

Syntax

```
InstSet = instoptembnd (CouponRate, Settle, Maturity,...
OptSpec, Strike, ExerciseDates, 'AmericanOpt',...
AmericanOpt, 'Period', Period, 'Basis', Basis,...
'EndMonthRule', EndMonthRule,'Face', Face, 'IssueDate',...
IssueDate, 'FirstCouponDate', FirstCouponDate,...
'LastCouponDate', LastCouponDate,'StartDate',StartDate)
InstSet = instoptembnd(InstSetOld, CouponRate,...)
[FieldList, ClassList, TypeString] = instoptembnd
```

Arguments

CouponRate	NINST-by-1 vector of decimal annual rate.
Settle	NINST-by-1 vector of settlement dates.
Maturity	NINST-by-1 vector of maturity dates.
OptSpec	NINST-by-1 vector of string values 'Call' or 'Put'.

For a European or Bermuda option

Strike	NINST-by-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 NaN's.
ExerciseDates	NINST-by-NSTRIKES matrix of exercise dates. Each row is the schedule for one option. For a European option, there is only one ExerciseDate on the option expiry date.
AmericanOpt	(Optional) NINST-by-1 vector of flags. AmericanOpt is 0 for each European or Bermuda option. The default is 0 if AmericanOpt is NaN or not entered.

For an American option

`Strike`	`NINST`-by-`1` vector of strike price values for each option.
`ExerciseDates`	`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 `ExerciseDate`.
`AmericanOpt`	`NINST`-by-`1` vector of flags. `AmericanOpt` is `1` for each American option. The `AmericanOpt` argument is required to invoke American exercise rules.
`Period`	(Optional) `NINST`-by-`1` matrix for coupons per year. The default value is `2`.
`Basis`	(Optional) Day-count basis of the instrument. `Basis` is a vector of integers with the following possible values: • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ICMA) • 9 = actual/360 (ICMA)

- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

`EndMonthRule` (Optional) `NINST`-by-`1` matrix for the end-of-month rule. This rule applies only when `Maturity` is an end-of-month date for a month having 30 or fewer days. When the value is `0`, the end-of-month rule is ignored, meaning that a bond's coupon payment date is always the same numerical day of the month. When the value is `1`, the end-of-month rule is set rule on (default), meaning that a bond's coupon payment date is always the last actual day of the month.

`IssueDate` (Optional) `NINST`-by-`1` matrix for the bond issue date.

`FirstCouponDate` (Optional) `NINST`-by-`1` matrix for an irregular first coupon date. 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) `NINST`-by-`1` matrix for an irregular last coupon date. Last coupon date of a bond before the maturity date. In the absence of a specified `FirstCouponDate`, a specified `LastCouponDate` determines the coupon structure of the bond. The coupon structure of a bond is truncated at the `LastCouponDate` regardless of where it falls and will be followed only by the bond's maturity cash flow date.

StartDate	(Optional) NINST-by-1 matrix (reserved input argument, currently unused) for date when a bond actually starts (the date from which a bond's cash flows can be considered). To make an instrument forward starting, specify this date as a future date. If StartDate is not explicitly specified, the effective start date is the settlement date.
Face	(Optional) NINST-by-1 matrix for the face value. The default value is 100.

Note 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 = instoptembnd (CouponRate, Settle,
Maturity,...OptSpec, Strike, ExerciseDates,
'AmericanOpt',...  AmericanOpt, 'Period',
Period, 'Basis', Basis,...  'EndMonthRule',
EndMonthRule,'Face', Face, 'IssueDate',...IssueDate,
'FirstCouponDate', FirstCouponDate,... 'LastCouponDate',
LastCouponDate,'StartDate',StartDate)
```

creates InstSet, a variable containing a collection of instruments.

Note instopembnd uses optional parameter name/value pairs such that, 'Name1', Value1, 'Name2', Value2, and so on, are a variable length list of name/value pairs.

Instruments are broken down by type and each type can have different data fields. Each stored data field has a row vector or string for each instrument. See `instget` for more information on the `InstSet` variable.

`InstSet = instoptembnd(InstSetOld, CouponRate,...)` adds `'OptEmBond'` instruments to an instrument variable.

`[FieldList, ClassList, TypeString] = instoptembnd` lists field metadata for the `'OptEmBond'` instrument.

`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 = 'OptEmBond'`.

Examples

To create a bond with embedded options with the following data:

```
Settle = 'jan-1-2007';
Maturity   = 'jan-1-2010';
CouponRate = 0.07;
OptSpec = 'call';
Strike= 100;
ExerciseDates= {'jan-1-2008' '01-Jan-2010'};
AmericanOpt=1;
Period = 1;

InstSet = instoptembnd(CouponRate, ...
Settle, Maturity, OptSpec, Strike,  ExerciseDates,'AmericanOpt', AmericanOpt, ...
'Period', Period);
```

To display the instrument:

```
instdisp(InstSet)
```

See Also

`instadd`, `instdisp`, `instget`

Purpose

Construct stock option

Syntax

```
InstSet = instoptstock(InstSet, OptSpec, Strike,
Settle, ExerciseDates)
InstSet = instoptstock(InstSet, OptSpec, Strike,
Settle, 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`instget` for more information on the `InstSet` 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`	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 `NaN`s. American option: `NINST`-by-`1` vector of strike price values for each option.
`Settle`	`NINST`-by-`1` vector of settlement 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

`InstSet = instoptstock(InstSet, OptSpec, Strike, Settle, ExerciseDates)` specifies a European or Bermuda option.

`InstSet = instoptstock(InstSet, OptSpec, Strike, Settle, 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

`InstSet`	Variable containing a collection of instruments. Instruments are classified by type; each type can have different data fields. The stored data field is a row vector or string for each instrument.
`FieldList`	String or number of fields (`NFIELDS`)-by-`1` cell array of strings listing the name of each data field to match with data values.
`DataList`	Number of values (`NVALUES`)-by-`M` array or `NFIELDS`-by-`1` cell array of acceptable data values for each field. Each row lists a data row value to search for in the corresponding `FieldList`. The number of columns is arbitrary and matching will ignore trailing `NaN`s or spaces.
`IndexSet`	(Optional) Number of instruments (`NINST`)-by-`1` vector restricting positions of instruments to check for matches. The default is all indices available in the instrument variable.
`TypeList`	(Optional) String or number of types (`NTYPES`)-by-`1` cell array of strings restricting instruments to match one of `TypeList` types. The default is all types in the instrument variable.

Argument value pairs can be entered in any order. The `InstSet` variable must be the first argument. `'FieldName'` and `'Data'` arguments must appear together or not at all. `'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')
```

```
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.

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
```

```
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, EndMonthRule)
[FieldList, ClassList, TypeString] = instswap
```

Arguments

`InstSet`	Instrument variable. This argument is specified only when adding a swap to an existing instrument set. See `instget` for more information on the `InstSet` variable.
`LegRate`	Number of instruments (`NINST`)-by-`2` matrix, with each row defined as: `[CouponRate Spread]` or `[Spread CouponRate]` `CouponRate` is the decimal annual rate. `Spread` is the number of basis points over the reference rate. The first column represents the receiving leg, while the second column represents the paying leg.
`Settle`	Settlement date. `NINST`-by-`1` vector of serial date numbers or date strings. `Settle` must be earlier than `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.

- 0 = actual/actual (default)
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

`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.

`EndMonthRule` (Optional) `NINST`-by-`1` vector representing the End-of-month rule. Default = `1`.

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, EndMonthRule)` 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'`.

Examples

Create a vanilla swap with the following data:

```
LegRate = [0.065, 0]
Settle = 'jan-1-2007';
Maturity = 'jan-1-2012';
LegReset = [1, 1];
Basis = 0
Principal = 100
LegType = [1, 0]

>> InstSet = instswap(LegRate, Settle, Maturity, LegReset, Basis, Principal, LegType);
```

View the swap instrument using `instdisp`:

```
>> instdisp(InstSet)
```

```
  Index Type LegRate    Settle         Maturity       LegReset Basis Principal LegType
  1     Swap [0.065  0] 01-Jan-2007    01-Jan-2012    1  1     0     100       [1  0]
```

See Also

hjmprice, instaddfield, instbond, instcap, instdisp, instfloor, intenvprice

Purpose

Construct swaption instrument

Syntax

```
InstSet = instswaption(OptSpec, Strike, ExerciseDates, ...
Spread, Settle, Maturity)
InstSet = instswaption(OptSpec, Strike, ExerciseDates, ...
Spread, Settle, Maturity, AmericanOpt, ...
SwapReset, Basis, Principal)
InstSet = instswaption(InstSetOld, OptSpec, Strike, ...
ExerciseDates, Spread, ...)
[FieldList, ClassList, TypeString] = instswaption;
```

Arguments

Fill unspecified entries in vectors with the value `NaN`. Only one data argument is required to create the instruments; the others may be omitted or passed as empty matrices [ ]. Type `[FieldList, ClassList] = instswaption` to see the classes. Dates can be input as serial date numbers or date strings.

`OptSpec`	`NINST`-by-1 cell array of strings `'call'` or `'put'`. A `'call'` swaption entitles the buyer to pay the fixed rate. A `'put'` swaption entitles the buyer to receive the fixed rate.
`Strike`	`NINST`-by-1 vector of strike swap rate values.

For a European option:

`ExerciseDates`	`NINST`-by-1 vector of exercise dates. Each row is the schedule for one option. For a European option, there is only one `ExerciseDate` on the option expiry date.
`AmericanOpt`	`NINST`-by-1 vector of flags. `AmericanOpt` is 0 for each European option. The default is 0 if `AmericanOpt` is `NaN` or not entered.

For an American option:

`ExerciseDates` `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 swap `Settle` and the single listed `ExerciseDate`.

`AmericanOpt` `NINST`-by-1 vector of flags. `AmericanOpt` is 1 for each American option. The `AmericanOpt` argument is required to invoke American exercise rules.

For an American or a European option:

`Spread` `NINST`-by-1 vector representing the number of basis points over the reference rate.

`Settle` `NINST`-by-1 vector of dates representing the settle date for each swap.

`Maturity` `NINST`-by-1 vector of dates representing the maturity date for each swap.

`SwapReset` (Optional) `NINST`-by-1 vector representing the reset frequency per year for the underlying swap. Default is 1.

`Basis` (Optional) Day-count basis of the instrument. A vector of integers.

- 0 = actual/actual (default)
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)

- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

`Principal` (Optional) `NINST`-by-`1` vector of the notional principal amounts. Default is 100.

Description

To specify a European option: `InstSet = instswaption(OptSpec, Strike, ExerciseDates, ...Spread, Settle, Maturity)`

To specify an American option: `InstSet = instswaption(OptSpec, Strike, ExerciseDates, ...Spread, Settle, Maturity, AmericanOpt, ...SwapReset, Basis, Principal)`

To add swaption instruments to an instrument variable: `InstSet = instswaption(InstSetOld, OptSpec, Strike, ...ExerciseDates, Spread, ...)`

To list field metadata for the swaption instrument: `[FieldList, ClassList, TypeString] = instswaption;`

Outputs:

`InstSet`	Variable containing a collection of instruments. Instruments are broken down by type and each type can have different data fields. Each stored data field has a row vector or string for each instrument. For more information on the `ISet` variable, see `instget`.
`FieldList`	`NFIELDS`-by-1 cell array of strings listing the name of each data field for this instrument type.
`ClassList`	`NFIELDS`-by-1 cell array of strings listing the data class of each field. The class determines how arguments will be parsed. Valid strings are `'dble'`, `'date'`, and `'char'`.
`TypeString`	String specifying the type of instrument added. `TypeString = 'Swaption'`.

Examples

Create two European swaption instruments with the following data:

```
OptSpec = {'Call'; 'Put'}
Strike = .05;
ExerciseDates = 'jan-1-2011';
Spread=0;
Settle = 'jan-1-2007';
Maturity = 'jan-1-2012';
AmericanOpt = 0;

OptSpec =

    'Call'
    'Put'

>> InstSet = instswaption(OptSpec, Strike, ExerciseDates, Spread, Settle, Maturity, ...
 AmericanOpt);
```

View the two European swaption instruments by using `instdisp`:

```
>> instdisp(InstSet)

Indx Type OptSpec Stke ExerDates Spread Settle Maturity AmerOpt SwpReset Basis Prinpal
1  Swaption Call  0.05  01-Jan-2011   0   01-Jan-2007   01-Jan-2012   0   1    0   100
2  Swaption Put   0.05  01-Jan-2011   0   01-Jan-2007   01-Jan-2012   0   1    0   100
```

See Also

instadd, instget, instdisp

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

`RateSpec`	A structure containing the properties of an interest-rate structure. See `intenvset` for information on creating `RateSpec`.
`ParameterName`	String indicating the parameter name to be accessed. The value of the named parameter is extracted from the structure `RateSpec`. It is sufficient to type only the leading characters that uniquely identify the parameter. Case is ignored for parameter names.

Description

`ParameterValue = intenvget(RateSpec,'ParameterName')` obtains the value of the named parameter `ParameterName` extracted from `RateSpec`.

Examples

Use `intenvset` to set the interest-rate structure.

```
RateSpec = intenvset('Rates', 0.05, 'StartDates',...
'20-Jan-2000', 'EndDates', '20-Jan-2001')
```

Now use `intenvget` to extract the values from `RateSpec`.

```
[R, RateSpec] = intenvget(RateSpec, 'Rates')

R =

    0.0500
RateSpec =

  FinObj: 'RateSpec'
```

```
    Compounding: 2
    Disc: 0.9518
    Rates: 0.0500
    EndTimes: 2
    StartTimes: 0
    EndDates: 730871
    StartDates: 730505
    ValuationDate: 730505
    Basis: 0
    EndMonthRule: 1
```

See Also

intenvset

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

intenvsens

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 `intenvset` for information on creating `RateSpec`.
`InstSet`	Variable containing a collection of instruments. Instruments are categorized by type; each type can have different data fields. The stored data field is a row vector or string for each instrument.

Description

`[Delta, Gamma, Price] = intenvsens(RateSpec, InstSet)` computes dollar prices and price sensitivities for instruments 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 ...
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
```

```
[Delta, Gamma] = intenvsens(ZeroRateSpec, ZeroInstSet)

Delta =

 -272.6403
 -347.4386
 -272.6403
   -1.0445
 -282.0405

Gamma =

  1.0e+003 *

    1.0298
    1.6227
    1.0298
```

```
    0.0033
    1.0596
```

See Also

hjmprice, hjmsens, instadd, intenvprice, intenvset

Purpose

Set properties of interest-rate 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 1 year.

`Compounding` = `365`

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

`Compounding` = `-1`

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

`Disc`	Number of points (`NPOINTS`) by number of curves (`NCURVES`) matrix of unit bond prices over investment intervals from `StartDates`, when the cash flow is valued, to `EndDates`, when the cash flow is received.
`Rates`	Number of points (`NPOINTS`) by number of curves (`NCURVES`) matrix of rates in decimal form. For example, 5% is `0.05` in `Rates`. `Rates` are the yields over investment intervals from `StartDates`, when the cash flow is valued, to `EndDates`, when the cash flow is received.
`EndDates`	`NPOINTS`-by-`1` vector or scalar of serial maturity dates ending the interval to discount over.
`StartDates`	`NPOINTS`-by-`1` vector or scalar of serial dates starting the interval to discount over. Default = `ValuationDate`. `StartDates` must be earlier than `EndDates`.
`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. • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European)

- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

`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.

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 before 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

True if input argument is financial structure type or financial object class

Syntax

```
IsFinObj = isafin(Obj, ClassName)
```

Arguments

`Obj`	Name of a financial structure.
`ClassName`	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, 'HJMFwdTree') returns True
```

See Also

`classfin`

Purpose

Price instruments using 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 `itttree` for information on creating the variable `ITTTree`.
`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 `derivset` 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 `NaN`s 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 ExerDates AmerOpt BarrSpec Barr Rebate Name Quantity
1   Barrier call    85   01-Jan-2006  31-Dec-2008    1       ui    115     0  Barrier1 1

IndxType OptSpec Strike Settle ExerDates 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(ITTTree, 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

ittsens

Purpose

Instrument sensitivities and prices using implied trinomial tree (ITT)

Syntax

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

Arguments

`ITTTree`	Implied trinomial stock tree. See `itttree` for information on creating the variable `ITTTree`.
`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 `derivset` containing derivative pricing options.

Description

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

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

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

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(ITTTree, ITTSubSet)


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 ittsens at 81

Delta =

    0.2387
   -0.4283
    0.3482

Gamma =

   0.0260
   0.0188
   0.0380
```

References

Chriss, Neil. and I. Kawaller, *Black-Scholes and Beyond: Options Pricing Models*, McGraw-Hill, 1996, pp. 308-312.

See Also

ittprice, itttree

itttimespec

Purpose

Specify time structure using implied trinomial tree (ITT)

Syntax

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

Arguments

`ValuationDate`	Scalar date marking the pricing date and first observation in the tree. Specify `ValuationDate` as a serial date number or date string.
`Maturity`	Scalar date marking the depth of the tree.
`NumPeriods`	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 four-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 implied trinomial stock tree

Syntax

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

Arguments

`StockSpec`	Stock specification. For more information, see `stockspec`.
`RateSpec`	Interest rate specification of the initial risk-free rate curve. For more information on declaring an interest rate variable, see `intenvset`.
`TimeSpec`	Tree time layout specification. Defines the observation dates of the implied trinomial tree. For more information on the tree structure, see `itttimespec`.
`StockOptSpec`	Option stock specification. For more information, see `stockoptspec`.

Description

`itttree(StockSpec, RateSpec, TimeSpec, StockOptSpec)` creates the `itttree` 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 the`StockSpec` 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: 'ITTTimeSpec'
    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';
    '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 In this example, the extrapolation warnings are turned on to display warnings on the Command Window. 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.

Use the following command to turn on extrapolation warnings:

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

Use the `StockSpec`, `RateSpec`, `TimeSpec`, and `StockOptSpec` structure to create an `ITTTree`.

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

```
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: 02-Jul-2006  Strike=60.7466
Option Type: 'put'   Maturity: 02-Jul-2006  Strike=50.0731
Option Type: 'put'   Maturity: 02-Jul-2006  Strike=41.3344
Option Type: 'call'   Maturity: 01-Jan-2007  Strike=73.8592
Option Type: 'call'   Maturity: 01-Jan-2007  Strike=60.8227
Option Type: 'put'   Maturity: 01-Jan-2007  Strike=50.1492
Option Type: 'put'   Maturity: 01-Jan-2007  Strike=41.4105
Option Type: 'put'   Maturity: 01-Jan-2007  Strike=34.2559
Option Type: 'call'   Maturity: 02-Jul-2007  Strike=88.8310
Option Type: 'call'   Maturity: 02-Jul-2007  Strike=72.9081
Option Type: 'call'   Maturity: 02-Jul-2007  Strike=59.8715
Option Type: 'put'   Maturity: 02-Jul-2007  Strike=49.1980
Option Type: 'put'   Maturity: 02-Jul-2007  Strike=40.4594
Option Type: 'put'   Maturity: 02-Jul-2007  Strike=33.3047
Option Type: 'put'   Maturity: 02-Jul-2007  Strike=27.4470
Option Type: 'call'   Maturity: 01-Jan-2008  Strike=107.2895
Option Type: 'call'   Maturity: 01-Jan-2008  Strike=87.8412
Option Type: 'call'   Maturity: 01-Jan-2008  Strike=71.9183
Option Type: 'call'   Maturity: 01-Jan-2008  Strike=58.8817
Option Type: 'put'   Maturity: 01-Jan-2008  Strike=48.2083
Option Type: 'put'   Maturity: 01-Jan-2008  Strike=39.4696
Option Type: 'put'   Maturity: 01-Jan-2008  Strike=32.3150
Option Type: 'put'   Maturity: 01-Jan-2008  Strike=26.4573
Option Type: 'put'   Maturity: 01-Jan-2008  Strike=21.6614

> In itttree>InterpOptPrices at 675
  In itttree at 277

ITTTree =

         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]}
```

See Also

intenvset, itttimespec, stockoptspec, stockspec

lookbackbycrr

Purpose

Price lookback option from CRR tree

Syntax

```
PriceTree = lookbackbycrr(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'`.
`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.

Description

`PriceTree = lookbackbycrr(CRRTree, OptSpec, Strike, Settle, ExerciseDates, AmericanOpt)` calculates the value of fixed- and floating-strike lookback options. 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 `[]`.

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

Note `lookbackbycrr` calculates values of fixed and floating strike lookback options. To compute the value of a floating strike lookback option, strike should be specified as `NaN`. Pricing of lookback options is done using Hull-White (1993). Consequently, for these options there are not unique prices on the tree nodes with the exception of the root 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`

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.

Description

`Price = lookbackbyeqp(EQPTree, OptSpec, Strike, ExerciseDates, AmericanOpt)` calculates the value of fixed- and floating-strike lookback options. 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 `[]`.

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

Note `lookbackbyeqp` calculates values of fixed and floating strike lookback options. To compute the value of a floating strike lookback option, strike should be specified as `NaN`. Pricing of lookback options is done using Hull-White (1993). Consequently, for these options there are not unique prices on the tree nodes with the exception of the root 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';
ExerciseDates = '01-Jan-2006';
```

```
Price = lookbackbyeqp(EQPTree, OptSpec, Strike, Settle, ...
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`

lookbackbyitt

Purpose

Price lookback option using 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 `ValuationDate` 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.

Description

`Price = lookbackbyitt(ITTTree, OptSpec, Strike, Settle, ExerciseDates, AmericanOpt)` calculates the value of fixed- and floating-strike lookback options. 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 `[]`.

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

Note `lookbackbyitt` calculates values of fixed and floating strike lookback options. To compute the value of a floating strike lookback option, strike should be specified as NaN. Pricing of lookback options is done using Hull-White (1993). Consequently, for these options there are not unique prices on the tree nodes with the exception of the root node.

Examples

Price a lookback option using an ITT equity 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.

```
OptSpec = 'Call';
Strike = 85;
Settle = '01-Jan-2006';
ExerciseDates = '01-Jan-2008';
```

```
Price = lookbackbyitt(ITTTree, 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`, `itttree`

Purpose

Calculate European rainbow option price on maximum of two risky assets using Stulz option pricing model

Syntax

```
Price = maxassetbystulz(RateSpec, StockSpec1, StockSpec2,
Settle, Maturity, OptSpec, Strike, Corr)
```

Arguments

`RateSpec`	The annualized, continuously compounded rate term structure. For information on the interest rate specification, see `intenvset`.
`StockSpec1`	Stock specification for asset 1. See `stockspec`.
`StockSpec2`	Stock specification for asset 2. See `stockspec`.
`Settle`	`NINST`-by-1 vector of settlement or trade dates.
`Maturity`	`NINST`-by-1 vector of maturity dates.
`OptSpec`	`NINST`-by-1 cell array of strings `'call'` or `'put'`.
`Strike`	`NINST`-by-1 vector of strike price values.
`Corr`	`NINST`-by-1 vector of correlation between the underlying asset prices.

Description

`Price = maxassetbystulz(RateSpec, StockSpec1, StockSpec2, Settle, Maturity, OptSpec, Strike, Corr)` computes rainbow option prices using the Stulz option pricing model.

`Price` is a `NINST`-by-1 vector of expected option prices.

Examples

Consider a European rainbow option that gives the holder the right to buy either $100,000 worth of an equity index at a strike price of 1000 (asset 1) or $100,000 of a government bond (asset 2) with a strike price of 100% of face value, whichever is worth more at the end of 12 months. On January 15, 2008, the equity index is trading at 950, pays a dividend

of 2% annually and has a return volatility of 22%. Also on January 15, 2008, the government bond is trading at 98, pays a coupon yield of 6%, and has a return volatility of 15%. The risk-free rate is 5%. Using this data, if the correlation between the rates of return is -0.5, 0, and 0.5, calculate the price of the European rainbow option.

Since the asset prices in this example are in different units, it is necessary to work in either index points (asset 1) or in dollars (asset 2). The European rainbow option allows the holder to buy the following: 100 units of the equity index at $1000 each (for a total of $100,000) or 1000 units of the government bonds at $100 each (for a total of $100,000). To convert the bond price (asset 2) to index units (asset 1), you must make the following adjustments:

- Multiply the strike price and current price of the government bond by 10 (1000/100).
- Multiply the option price by 100, considering that there are 100 equity index units in the option.

Once these adjustments are introduced, the strike price is the same for both assets ($1000).

Create the `RateSpec`:

```
Settle = 'Jan-15-2008';
Maturity = 'Jan-15-2009';
Rates = 0.05;
Basis = 1;

RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle,...
'EndDates', Maturity, 'Rates', Rates, 'Compounding', -1, 'Basis', Basis);
```

Create the two `StockSpec` definitions:

```
AssetPrice1 = 950;   % Asset 1 => Equity index
AssetPrice2 = 980;   % Asset 2 => Government bond
Sigma1 = 0.22;
Sigma2 = 0.15;
```

```
Div1 = 0.02;
Div2 = 0.06;

StockSpec1 = stockspec(Sigma1, AssetPrice1, 'continuous', Div1);
StockSpec2 = stockspec(Sigma2, AssetPrice2, 'continuous', Div2);
```

Calculate the price of the options for different correlation levels:

```
Strike = 1000 ;
Corr = [-0.5; 0; 0.5];
OptSpec = 'call';

Price = maxassetbystulz(RateSpec, StockSpec1, StockSpec2,...
Settle, Maturity, OptSpec, Strike, Corr)

Price =

  111.6683
  103.7715
   92.4412
```

These are the prices of one unit. This means that the premium is 11166.83, 10377.15, and 9244.12 (for 100 units).

See Also

`intenvset`, `maxassetsensbystulz`, `minassetbystulz`, `stockspec`

maxassetsensbystulz

Purpose

Calculate European rainbow option prices and sensitivities on maximum of two risky assets using Stulz pricing model

Syntax

```
PriceSens = maxassetsensbystulz(RateSpec, StockSpec1,
StockSpec2, Settle, Maturity, OptSpec, Strike, Corr)
PriceSens = maxassetsensbystulz(RateSpec, StockSpec1,
StockSpec2, Settle, Maturity, OptSpec, Strike, Corr, OutSpec)
```

Arguments

`RateSpec`	The annualized, continuously compounded rate term structure. For information on the interest rate specification, see `intenvset`.
`StockSpec1`	Stock specification for asset 1. See `stockspec`.
`StockSpec2`	Stock specification for asset 2. See `stockspec`.
`Settle`	`NINST`-by-`1` vector of settlement or trade dates.
`Maturity`	`NINST`-by-`1` vector of maturity dates.
`OptSpec`	`NINST`-by-`1` cell array of strings `'call'` or `'put'`.
`Strike`	`NINST`-by-`1` vector of strike price values.
`Corr`	`NINST`-by-`1` vector of correlation between the underlying asset prices.
`OutSpec`	(Optional) All optional inputs are specified as matching parameter name/value pairs. The parameter name is specified as a character string, followed by the corresponding parameter value. You can specify parameter name/value pairs in any order. Names are case-insensitive and partial string matches are allowed provided no ambiguities exist. Valid parameter names are: • `NOUT`-by-`1` or `1`-by-`NOUT` cell array of strings indicating the nature and order of the outputs for the function. Possible values are `'Price'`,

'Delta', 'Gamma', 'Vega', 'Lambda', 'Rho', 'Theta', or 'All'.

For example, OutSpec = {'Price'; 'Lamba'; 'Rho'} specifies that the output should be Price, Lambda, and Rho, in that order.

To invoke from a function: [Price, Lambda, Rho] = maxassetsensbystulz(..., 'OutSpec', {'Price', 'Lamba', 'Rho'})

OutSpec = {'All'} specifies that the output should be Delta, Gamma, Vega, Lambda, Rho, Theta, and Price, in that order. This is the same as specifying OutSpec as OutSpec = {'Delta', 'Gamma', 'Vega', 'Lambda', 'Rho', 'Theta', 'Price'};.

- Default is OutSpec = {'Price'}.

Description

PriceSens = maxassetsensbystulz(RateSpec, StockSpec1, StockSpec2, Settle, Maturity, OptSpec, Strike, Corr) computes rainbow option prices using the Stulz option pricing model.

PriceSens = maxassetsensbystulz(RateSpec, StockSpec1, StockSpec2, Settle, Maturity, OptSpec, Strike, Corr, OutSpec) computes rainbow option prices and sensitivities using the Stulz option pricing model.

PriceSens is a NINST-by-1 or NINST-by-2 vector of expected prices and sensitivities values.

Examples

Consider a European rainbow option that gives the holder the right to buy either $100,000 of an equity index at a strike price of 1000 (asset 1) or $100,000 of a government bond (asset 2) with a strike price of 100% of face value, whichever is worth more at the end of 12 months. On January 15, 2008, the equity index is trading at 950, pays a dividend of 2% annually, and has a return volatility of 22%. Also on January 15,

2008, the government bond is trading at 98, pays a coupon yield of 6%, and has a return volatility of 15%. The risk-free rate is 5%. Using this data, calculate the price and sensitivity of the European rainbow option if the correlation between the rates of return is -0.5, 0, and 0.5.

Since the asset prices in this example are in different units, it is necessary to work in either index points (for asset 1) or in dollars (for asset 2). The European rainbow option allows the holder to buy the following: 100 units of the equity index at $1000 each (for a total of $100,000) or 1000 units of the government bonds at $100 each (for a total of $100,000). To convert the bond price (asset 2) to index units (asset 1), uou must make the following adjustments:

- Multiply the strike price and current price of the government bond by 10 (1000/100).
- Multiply the option price by 100, considering that there are 100 equity index units in the option.

Once these adjustments are introduced, the strike price is the same for both assets ($1000).

Create the `RateSpec`:

```
Settle = 'Jan-15-2008';
Maturity = 'Jan-15-2009';
Rates = 0.05;
Basis = 1;

RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle,...
'EndDates', Maturity, 'Rates', Rates, 'Compounding', -1, 'Basis', Basis);
```

Create the two `StockSpec` definitions:

```
AssetPrice1 = 950;   % Asset 1 => Equity index
AssetPrice2 = 980;   % Asset 2 => Government bond
Sigma1 = 0.22;
Sigma2 = 0.15;
Div1 = 0.02;
```

```
Div2 = 0.06;

StockSpec1 = stockspec(Sigma1, AssetPrice1, 'continuous', Div1);
StockSpec2 = stockspec(Sigma2, AssetPrice2, 'continuous', Div2);
```

Calculate the price and delta for different correlation levels:

```
Strike = 1000 ;
Corr = [-0.5; 0; 0.5];
OutSpec = {'price'; 'delta'};
[Price, Delta] = maxassetsensbystulz(RateSpec, StockSpec1, StockSpec2,...
Settle, Maturity, OptSpec, Strike, Corr,'OutSpec', OutSpec)

Price =

  111.6683
  103.7715
   92.4412

Delta =

    0.4594    0.3698
    0.4292    0.3166
    0.4053    0.2512
```

The output `Delta` has two columns: the first column represents the `Delta` with respect to the equity index (asset 1), and the second column represents the `Delta` with respect to the government bond (asset 2). The value 0.4595 represents `Delta` with respect to one unit of the equity index. Since there are 100 units of the equity index, the overall `Delta` would be 45.94 (100 * 0.4594) for a correlation level of -0.5. To calculate the `Delta` with respect to the government bond, remember that an adjusted price of 980 was used instead of 98. Therefore, for example, the `Delta` with respect to government bond, for a correlation of 0.5 would be 251.2 (0.2512 * 100 * 10).

See Also

intenvset, maxassetbystulz, stockspec

Purpose

Calculate European rainbow option prices on minimum of two risky assets using Stulz option pricing model

Syntax

```
Price = minassetbystulz(RateSpec, StockSpec1, StockSpec2,
Settle, Maturity, OptSpec, Strike, Corr)
```

Arguments

`RateSpec`	The annualized, continuously compounded rate term structure. For information on the interest rate specification, see `intenvset`.
`StockSpec1`	Stock specification for asset 1. See `stockspec`.
`StockSpec2`	Stock specification for asset 2. See `stockspec`.
`Settle`	`NINST`-by-`1` vector of settlement or trade dates.
`Maturity`	`NINST`-by-`1` vector of maturity dates.
`OptSpec`	`NINST`-by-`1` cell array of strings `'call'` or `'put'`.
`Strike`	`NINST`-by-`1` vector of strike price values.
`Corr`	`NINST`-by-`1` vector of correlation between the underlying asset prices.

Description

`Price = minassetbystulz(RateSpec, StockSpec1, StockSpec2, Settle, Maturity, OptSpec, Strike, Corr)` computes option prices using the Stulz option pricing model.

`Price` is a `NINST`-by-`1` vector of expected option prices.

Examples

Consider a European rainbow put option that gives the holder the right to sell either stock A or stock B at a strike of 50.25, whichever has the lower value on the expiration date May 15, 2009. On November 15, 2008, stock A is trading at 49.75 with a continuous annual dividend yield of 4.5% and has a return volatility of 11%. Stock B is trading at 51 with a continuous dividend yield of 5% and has a return volatility

of 16%. The risk-free rate is 4.5%. Using this data, if the correlation between the rates of return is -0.5, 0, and 0.5, calculate the price of the minimum of two assets that are European rainbow put options.

Create the `RateSpec`:

```
Settle = 'Nov-15-2008';
Maturity = 'May-15-2009';
Rates = 0.045;
Basis = 1;

RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle,...
'EndDates', Maturity, 'Rates', Rates, 'Compounding', -1, 'Basis', Basis);
```

Create the two `StockSpec` definitions:

```
AssetPriceA = 49.75;
AssetPriceB = 51;
SigmaA = 0.11;
SigmaB = 0.16;
DivA = 0.045;
DivB = 0.05;

StockSpecA = stockspec(SigmaA, AssetPriceA, 'continuous', DivA);
StockSpecB = stockspec(SigmaB, AssetPriceB, 'continuous', DivB);
```

Compute the price of the options for different correlation levels:

```
Strike = 50.25;
Corr = [-0.5;0;0.5];
OptSpec = 'put';

Price = minassetbystulz(RateSpec, StockSpecA, StockSpecB, Settle,...
Maturity, OptSpec, Strike, Corr)

Price =

    3.4320
```

```
3.1384
2.7694
```

The values 3.43, 3.14, and 2.77 are the price of the European rainbow put options with a correlation level of -0.5, 0, and 0.5 respectively.

See Also

`intenvset`, `maxassetbystulz`, `minassetsensbystulz`, `stockspec`

minassetsensbystulz

Purpose

Calculate European rainbow option prices and sensitivities on minimum of two risky assets using Stulz pricing model

Syntax

```
PriceSens = minassetsensbystulz(RateSpec, StockSpec1,
StockSpec2, Settle, Maturity, OptSpec, Strike, Corr)
PriceSens = minassetsensbystulz(RateSpec, StockSpec1,
StockSpec2, Settle, Maturity, OptSpec, Strike, Corr, OutSpec)
```

Arguments

`RateSpec`	The annualized continuously compounded rate term structure. For information on the interest rate specification, see `intenvset`.
`StockSpec1`	Stock specification for asset 1. See `stockspec`.
`StockSpec2`	Stock specification for asset 2. See `stockspec`.
`Settle`	`NINST`-by-`1` vector of settlement or trade dates.
`Maturity`	`NINST`-by-`1` vector of maturity dates.
`OptSpec`	`NINST`-by-`1` cell array of strings `'call'` or `'put'`.
`Strike`	`NINST`-by-`1` vector of strike price values.
`Corr`	`NINST`-by-`1` vector of correlation between the underlying asset prices.
`OutSpec`	(Optional) All optional inputs are specified as matching parameter name/value pairs. The parameter name is specified as a character string, followed by the corresponding parameter value. You can specify parameter name/value pairs in any order. Names are case-insensitive and partial string matches are allowed provided no ambiguities exist. Valid parameter names are: • `NOUT`-by-`1` or `1`-by-`NOUT` cell array of strings indicating the nature and order of the outputs for the function. Possible values are `'Price'`,

'Delta', 'Gamma', 'Vega', 'Lambda', 'Rho', 'Theta', or 'All'.

For example, OutSpec = {'Price'; 'Lamba'; 'Rho'} specifies that the output should be Price, Lambda, and Rho, in that order.

To invoke from a function: [Price, Lambda, Rho] = minassetsensbystulz(..., 'OutSpec', {'Price', 'Lamba', 'Rho'})

OutSpec = {'All'} specifies that the output should be Delta, Gamma, Vega, Lambda, Rho, Theta, and Price, in that order. This is the same as specifying OutSpec as OutSpec = {'Delta', 'Gamma', 'Vega', 'Lambda', 'Rho', 'Theta', 'Price'};.

- Default is OutSpec = {'Price'}.

Description

PriceSens = minassetsensbystulz(RateSpec, StockSpec1, StockSpec2, Settle, Maturity, OptSpec, Strike, Corr) computes rainbow option prices using the Stulz option pricing model.

PriceSens = minassetsensbystulz(RateSpec, StockSpec1, StockSpec2, Settle, Maturity, OptSpec, Strike, Corr, OutSpec) computes rainbow option prices and sensitivities using the Stulz option pricing model.

PriceSens is a NINST-by-1 or NINST-by-2 vector of expected prices and sensitivities.

Examples

Consider a European rainbow put option that gives the holder the right to sell either stock A or stock B at a strike of 50.25, whichever has the lower value on the expiration date May 15, 2009. On November 15, 2008, stock A is trading at 49.75 with a continuous annual dividend yield of 4.5% and has a return volatility of 11%. Stock B is trading at 51 with a continuous dividend yield of 5% and has a return volatility

of 16%. The risk-free rate is 4.5%. Using this data, if the correlation between the rates of return is -0.5, 0, and 0.5, calculate the price and sensitivity of the minimum of two assets that are European rainbow put options.

Create the `RateSpec`:

```
Settle = 'Nov-15-2008';
Maturity = 'May-15-2009';
Rates = 0.045;
Basis = 1;

RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle,...
'EndDates', Maturity, 'Rates', Rates, 'Compounding', -1, 'Basis', Basis);
```

Create the two `StockSpec` definitions:

```
AssetPriceA = 49.75;
AssetPriceB = 51;
SigmaA = 0.11;
SigmaB = 0.16;
DivA = 0.045;
DivB = 0.05;

StockSpecA = stockspec(SigmaA, AssetPriceA, 'continuous', DivA);
StockSpecB = stockspec(SigmaB, AssetPriceB, 'continuous', DivB);
```

Calculate price and delta for different correlation levels:

```
Strike = 50.25;
Corr = [-0.5;0;0.5];
OutSpec = {'Price'; 'delta'};
[P, D] = minassetsensbystulz(RateSpec, StockSpecA, StockSpecB,...
Settle, Maturity, OptSpec, Strike, Corr, 'OutSpec', OutSpec)

P =

    3.4320
```

```
    3.1384
    2.7694

D =

   -0.4183   -0.3496
   -0.3746   -0.3189
   -0.3304   -0.2905
```

The output `Delta` has two columns: the first column represents the `Delta` with respect to the stock A (asset 1), and the second column represents the `Delta` with respect to the stock B (asset 2). The value 0.4183 represents `Delta` with respect to the stock A for a correlation level of -0.5. The `Delta` with respect to stock B, for a correlation of zero is -0.3189.

See Also

`intenvset`, `minassetbystulz`, `stockspec`

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`.

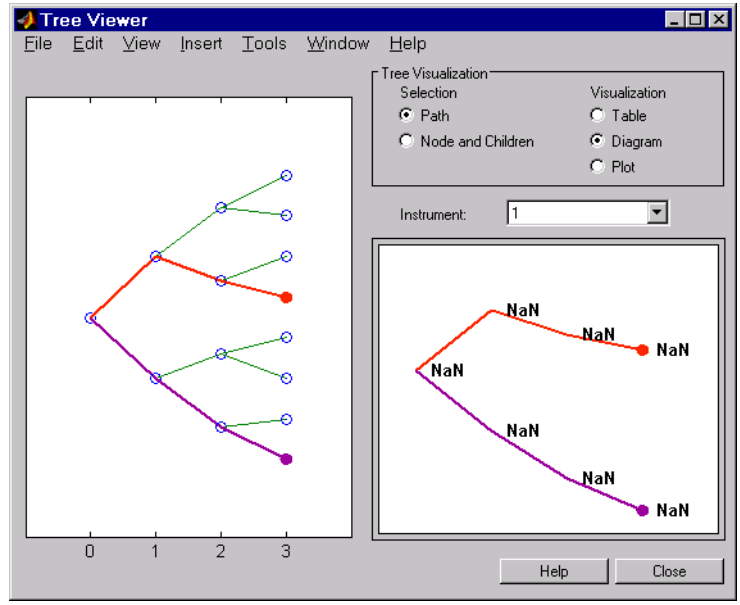
Description

`[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

`NumLevels`	Number of time levels of the tree.
`NumPos`	1-by-`NUMLEVELS` vector containing the length of the state vectors in each time level.
`NodeVal`	(Optional) Initial value at each node of the tree. Default = `NaN`.
`IsPriceTree`	(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

`NumLevels`	Number of time levels of the tree.
`NumPos`	1-by-`NUMLEVELS` vector containing the length of the state vectors in each time level.
`NumStates`	1-by-`NUMLEVELS` vector containing the number of state vectors in each time level.
`NodeVal`	(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(BDTTree)
```

Arguments

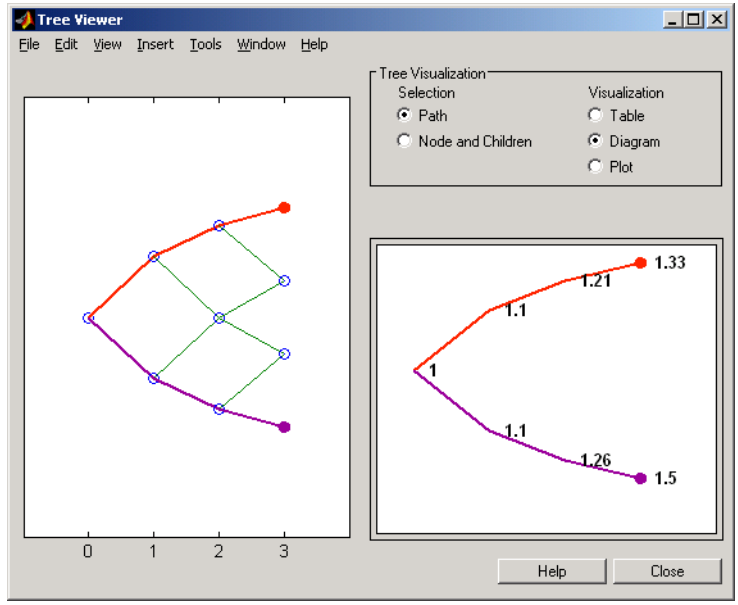
`BDTTree`	Interest-rate tree structure created by `bdttree`.

Description

`MMktTree = mmktbybdt(BDTTree)` creates a money-market tree from an interest-rate tree structure created by `bdttree`.

Examples

```
load deriv.mat;
MMktTree = mmktbybdt(BDTTree);
treeviewer(MMktTree)
```



See Also

bdttree

mmktbyhjm

Purpose

Create money-market tree from HJM interest-rate tree

Syntax

```
MMktTree = mmktbyhjm(HJMTree)
```

Arguments

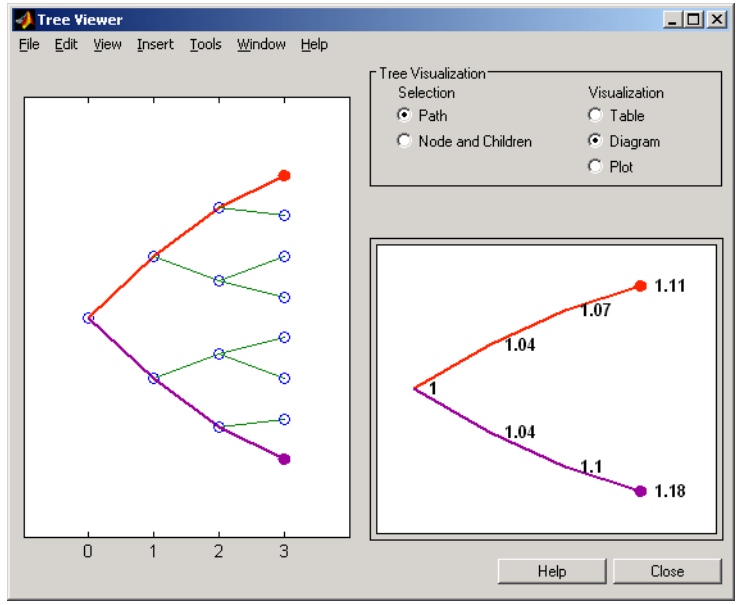
`HJMTree`	Forward-rate tree structure created by `hjmtree`.

Description

`MMktTree = mmktbyhjm(HJMTree)` creates a money-market tree from a forward-rate tree structure created by `hjmtree`.

Examples

```
load deriv.mat;
MMktTree = mmktbyhjm(HJMTree);
treeviewer(MMktTree)
```



See Also

hjmtree

optbndbybdt

Purpose

Price bond option from BDT interest-rate tree

Syntax

```
[Price, PriceTree] = optbndbybdt(BDTTree, OptSpec, Strike,
ExerciseDates, AmericanOpt, CouponRate, Settle, Maturity,
Period,  Basis, EndMonthRule, IssueDate, FirstCouponDate,
LastCouponDate, StartDate, Face, Options)
```

Arguments

`BDTTree`	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 `NaN`s. For an American option: `NINST`-by-`1` vector of strike price values for each option.

`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.
`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 `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. • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ICMA) • 9 = actual/360 (ICMA) • 10 = actual/365 (ICMA) • 11 = 30/360E (ICMA) • 12 = actual/actual (ISDA) • 13 = BUS/252
`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`	(Optional) Date when a bond actually starts (the date from which a bond cash flow is considered). To make an instrument forward-starting, specify this date as a future date. If you do not specify `StartDate`, the effective start date is the `Settle` date.
`Face`	(Optional) Face value. Default is `100`.
`Options`	(Optional) Derivatives pricing options structure created with `derivset`.

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

Description

`[Price, PriceTree] = optbndbybdt(BDTTree, OptSpec, Strike, ExerciseDates, AmericanOpt, CouponRate, Settle, Maturity, Period, Basis, EndMonthRule, IssueDate, FirstCouponDate, LastCouponDate, StartDate, Face, Options)` computes the price of a bond option from a BDT interest-rate tree.

`Price` is an `NINST`-by-1 matrix of expected prices at time 0.

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

Examples

Example 1. Using the BDT interest-rate tree in the `deriv.mat` file, price a European call option on a 10% bond with a strike of 95. The exercise date for the option is Jan. 01, 2002. The settle date for the bond is Jan. 01, 2000, and the maturity date is Jan. 01, 2003.

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

```
load deriv.mat;
```

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

```
Price = optbndbybdt(BDTTree,'Call',95,'01-Jan-2002',...
'0','0.10','01-Jan-2000','01-Jan-2003','1')

Price =

    1.7657
```

Example 2. Now use `optbndbybdt` to compute the price of a put option on the same bond.

```
Price = optbndbybdt(BDTTree,'Put',95,'01-Jan-2002',...
'0','0.10','01-Jan-2000','01-Jan-2003','1')

Price =

    0.5740
```

See Also

`bdtprice`, `bdttree`, `instoptbnd`

Purpose

Price bond option 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`	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 `NaN`s. For an American option: `NINST`-by-`1` vector of strike price values for each option.

`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.
`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 `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. • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ICMA) • 9 = actual/360 (ICMA) • 10 = actual/365 (ICMA) • 11 = 30/360E (ICMA) • 12 = actual/actual (ISDA) • 13 = BUS/252
`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`	(Optional) Date when a bond actually starts (the date from which a bond cash flow is considered). To make an instrument forward-starting, specify this date as a future date. If you do not specify `StartDate`, the effective start date is the `Settle` date.
`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 `optbondbybk` 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: OptBonds are valued at Tree ValuationDate rather than Settle
> In optbndbytrintree at 43
  In optbndbybk at 88
Warning: Not all cash flows are aligned with the tree. Result will be
approximated.
> In optbndbytrintree at 151
  In optbndbybk at 88

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: OptBonds are valued at Tree ValuationDate rather than Settle
```

```
> In optbndbytrintree at 43
  In optbndbybk at 88
Warning: Not all cash flows are aligned with the tree. Result will be
approximated.
> In optbndbytrintree at 151
  In optbndbybk at 88

Price =

    0.0272
```

See Also

bkprice, bktree, instoptbnd

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 `NaN`s. For an American option: `NINST`-by-`1` vector of strike price values for each option.

`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.
`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 `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.

- 0 = actual/actual (default)
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

`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`	(Optional) Date when a bond actually starts (the date from which a bond cash flow is considered). To make an instrument forward-starting, specify this date as a future date. If you do not specify `StartDate`, the effective start date is the `Settle` date.
`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 `optbondbyhjm` to compute the price of the option.

```
Price = optbndbyhjm(HJMTree,'Call',96,'01-Jan-2003',...
'0','0.04','01-Jan-2000','01-Jan-2004')
Warning: Not all cash flows are aligned with the tree. Result will
be approximated.

Price =

    2.2410
```

See Also

`hjmprice`, `hjmtree`, `instoptbnd`

optbndbyhw

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`	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 `NaN`s. For an American option: `NINST`-by-`1` vector of strike price values for each option.

`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.
`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 `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. • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ICMA) • 9 = actual/360 (ICMA) • 10 = actual/365 (ICMA) • 11 = 30/360E (ICMA) • 12 = actual/actual (ISDA) • 13 = BUS/252
`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 before 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`	(Optional) Date when a bond actually starts (the date from which a bond cash flow is considered). To make an instrument forward-starting, specify this date as a future date. If you do not specify `StartDate`, the effective start date is the `Settle` date.
`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 `optbondbyhw` 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 `optbondbyhw` 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

optembndbybdt

Purpose

Price bonds with embedded options by Black-Derman-Toy interest rate tree

Syntax

```
[Price, PriceTree] = optembndbybdt(BDTTree, CouponRate,
Settle, Maturity, OptSpec, Strike, ExerciseDates,
'Name1', Value1, 'Name2', Value2, ...)
```

Arguments

`BDTTree`	Interest-rate tree structure created by `bdttree`.
`CouponRate`	`NINST`-by-1 matrix for the decimal annual rate.
`Settle`	`NINST`-by-1 matrix for the settlement date. A vector of serial date numbers or date strings. `Settle` must be earlier than `Maturity`.
`Maturity`	`NINST`-by-1 matrix for the maturity date. A vector of serial date numbers or date strings.
`OptSpec`	`NINST`-by-1 cell array of strings `'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 `NaN`s. For an American option: `NINST`-by-1 vector of strike price values for each option.

`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.
`'Name1', 'Value1' 'Name2', 'Value2'...`	(Optional) The name/value pairs are a variable length list of parameters. All optional inputs are specified as matching parameter name/value pairs. The parameter name is specified as a character string, followed by the corresponding parameter value. Parameter name/value pairs may be specified in any order; names are case insensitive and partial string matches are allowed provided no ambiguities exist. Valid parameter names are as follows: • `AmericanOpt` is a `NINST`-by-`1` matrix for flags options: `0` (European/Bermuda) or `1` (American). Default is `0`. • `Period` is a `NINST`-by-`1` matrix for coupons per year. The default value is `2`.

- `Basis` is a day-count basis of the instrument. `Basis` is a vector of integers with the following possible values:
  - 0 = actual/actual (default)
  - 1 = 30/360 (SIA)
  - 2 = actual/360
  - 3 = actual/365
  - 4 = 30/360 (BMA)
  - 5 = 30/360 (ISDA)
  - 6 = 30/360 (European)
  - 7 = actual/365 (Japanese)
  - 8 = actual/actual (ICMA)
  - 9 = actual/360 (ICMA)
  - 10 = actual/365 (ICMA)
  - 11 = 30/360E (ICMA)
  - 12 = actual/actual (ISDA)
  - 13 = BUS/252
- `EndMonthRule` is a `NINST`-by-`1` matrix for the end-of-month rule. This rule applies only when `Maturity` is an end-of-month date for a month having 30 or fewer days. When the value is `0` the end-of-month rule is ignored; this means that a bond's coupon payment date is always the same numerical day of the month. Use `1` to set the rule on; this is the default value and means that a bond's coupon payment date is always the last actual day of the month.

- `IssueDate` is a `NINST`-by-`1` matrix for the bond issue date.
- `FirstCouponDate` is a `NINST`-by-`1` matrix for an irregular first coupon date. This is the 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` is a `NINST`-by-`1` matrix for an irregular last coupon date. This is the last coupon date of a bond before the maturity date. In the absence of a specified `FirstCouponDate`, a specified `LastCouponDate` determines the coupon structure of the bond. The coupon structure of a bond is truncated at the `LastCouponDate` regardless of where it falls and will be followed only by the bond's maturity cash flow date.
- `StartDate` is a `NINST`-by-`1` matrix for date when a bond actually starts (the date from which a bond cash flow is considered). To make an instrument forward-starting, specify this date as a future date. If you do not specify `StartDate`, the effective start date is the `Settle` date.
- `Face` is a `NINST`-by-`1` matrix for the face value. The default value is `100`.
- `Options` is a derivatives pricing options structure created with `derivset`.

Note The `Settle` date for every bond with an embedded option is set to the `ValuationDate` of the BDT tree; the bond's argument for `Settle` date is ignored.

Description

`[Price, PriceTree] = optembndbybdt(BDTTree, CouponRate,Settle, Maturity, OptSpec, Strike, ExerciseDates,'Name1', Value1, 'Name2', Value2, ...)` prices bonds with embedded options using 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

To price a callable bond using the BDT model, create a BDTTree with the following data:

```
ZeroRates = [ 0.035;0.04;0.045];
Compounding = 1;
StartDates = ['jan-1-2007';'jan-1-2008';'jan-1-2009'];
EndDates   = ['jan-1-2008';'jan-1-2009';'jan-1-2010'];
ValuationDate = 'jan-1-2007';
```

Create a `RateSpec`:

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

Specify a `VolSpec`:

```
Volatility = 0.10 * ones (3,1);
VolSpec = bdtvolspec(ValuationDate, EndDates, Volatility);
```

Specify a `TimeSpec`:

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

Build the BDTTree:

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

To compute the price of an American callable bond that pays a 5.25% annual coupon, matures in Jan-1-2010, and is callable on Jan-1-2008 and 01-Jan-2010:

```
BondSettlement = 'jan-1-2007';
BondMaturity   = 'jan-1-2010';
CouponRate = 0.0525;
Period = 1;
OptSpec = 'call';
Strike = [100];
ExerciseDates = {'jan-1-2008' '01-Jan-2010'};
AmericanOpt = 1;

PriceCallBond = optembndbybdt(BDTTree, CouponRate, BondSettlement, BondMaturity,...
OptSpec, Strike, ExerciseDates, 'Period', 1,'AmericanOp', 1)

PriceCallBond =

101.4750
```

See Also

`bdtprice`, `bdttree`, `instoptembnd`

optembndbybk

Purpose

Price bonds with embedded options by Black-Karasinski interest-rate tree

Syntax

```
[Price, PriceTree] = optembndybk(BKTree, CouponRate,
Settle, Maturity, OptSpec, Strike, Strike, ExerciseDates,
'Name1', Value1, 'Name2', Value2, ...)
```

Arguments

`BKTree`	Interest-rate tree structure created by `bktree`.
`CouponRate`	`NINST`-by-1 matrix for the decimal annual rate.
`Settle`	`NINST`-by-1 matrix for the settlement date. A vector of serial date numbers or date strings. `Settle` must be earlier than `Maturity`.
`Maturity`	`NINST`-by-1 matrix for the maturity date. A vector of serial date numbers or date strings.
`OptSpec`	`NINST`-by-1 cell array of strings `'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 `NaN`s. For an American option: `NINST`-by-1 vector of strike price values for each option.

`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.
`'Name1', Value1, 'Name2', Value2 ...`	(Optional) The name/value pairs are a variable length list of parameters. All optional inputs are specified as matching parameter name/value pairs. The parameter name is specified as a character string, followed by the corresponding parameter value. Parameter name/value pairs may be specified in any order; names are case insensitive and partial string matches are allowed provided no ambiguities exist. Valid parameter names are as follows: • `AmericanOpt` is a `NINST`-by-`1` matrix for flags options: `0` (European/Bermuda) or `1` (American). Default is `0`. • `Period` is a `NINST`-by-`1` matrix for coupons per year. Default is `2`.

- `Basis` is a day-count basis of the instrument. `Basis` is a vector of integers with the following supported values:
  - 0 = actual/actual (default)
  - 1 = 30/360 (SIA)
  - 2 = actual/360
  - 3 = actual/365
  - 4 = 30/360 (BMA)
  - 5 = 30/360 (ISDA)
  - 6 = 30/360 (European)
  - 7 = actual/365 (Japanese)
  - 8 = actual/actual (ICMA)
  - 9 = actual/360 (ICMA)
  - 10 = actual/365 (ICMA)
  - 11 = 30/360E (ICMA)
  - 12 = actual/actual (ISDA)
  - 13 = BUS/252
- `EndMonthRule` is a `NINST`-by-`1` matrix for the end-of-month rule. This rule applies only when `Maturity` is an end-of-month date for a month having 30 or fewer days. When the value is `0`, the end-of-month rule is ignored, meaning that a bond's coupon payment date is always the same numerical day of the month. When the value is `1`, the end-of-month rule is set on (default), meaning that a bond's coupon payment date is always the last actual day of the month.

- `IssueDate` is a `NINST`-by-`1` matrix for the bond issue date.
- `FirstCouponDate` is a `NINST`-by-`1` matrix for an irregular first coupon date. 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` is a `NINST`-by-`1` matrix for an irregular last coupon date. Last coupon date of a bond before the maturity date. In the absence of a specified `FirstCouponDate`, a specified `LastCouponDate` determines the coupon structure of the bond. The coupon structure of a bond is truncated at the `LastCouponDate` regardless of where it falls and will be followed only by the bond's maturity cash flow date.
- `StartDate` is a `NINST`-by-`1` matrix for date when a bond actually starts (the date from which a bond cash flow is considered). To make an instrument forward-starting, specify this date as a future date. If you do not specify `StartDate`, the effective start date is the `Settle` date.
- `Face` is a `NINST`-by-`1` matrix for the face value. The default value is `100`.
- `Options` is a derivatives pricing options structure created with `derivset`.

Note The `Settle` date for every bond with embedded option is set to the `ValuationDate` of the BKTree; the bond's argument for `Settle` date is ignored.

Description

`[Price, PriceTree] = optembndybk(BKTree, CouponRate, Settle, Maturity, OptSpec, Strike, Strike, ExerciseDates, 'Name1', Value1, 'Name2', Value2, ...)` prices bonds with embedded options by a BK 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

Create a `BKTree` with the following data:

```
ZeroRates = [ 0.035;0.04;0.045];
Compounding = 1;
StartDates = ['jan-1-2007';'jan-1-2008';'jan-1-2009'];
EndDates   = ['jan-1-2008';'jan-1-2009';'jan-1-2010'];
ValuationDate = 'jan-1-2007';
```

Create a `RateSpec`:

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

Specify a `TimeSpec`:

```
BKTimeSpec = bktimespec(ValuationDate, EndDates, Compounding);
```

Specify a `VolSpec`:

```
VolDates = ['jan-1-2008';'jan-1-2009';'jan-1-2010'];
VolCurve = 0.01;
AlphaDates = 'jan-1-2010';
AlphaCurve = 0.1;
BKVolSpec = bkvolspec(ValuationDate, VolDates, VolCurve, AlphaDates, AlphaCurve);
```

Build a `BKTree`:

```
BKTree = bktree(BKVolSpec, RateSpec, BKTimeSpec);
```

To compute the price of an American puttable bond that pays an annual coupon of 5.25% , matures on January 1, 2010, and is callable on January 1, 2008 and January 1, 2010:

```
BondSettlement = 'jan-1-2007';
BondMaturity   = 'jan-1-2010';
CouponRate = 0.0525;
Period = 1;
OptSpec = 'put';
Strike = [100];
ExerciseDates = {'jan-1-2008' '01-Jan-2010'};
AmericanOpt = 1;

PricePutBondBK = optembndbybk(BKTree, CouponRate, BondSettlement, BondMaturity,...
OptSpec, Strike, ExerciseDates,'Period', 1, 'AmericanOpt', 1)

PricePutBondBK =

102.3820
```

See Also

`bkprice`, `bktree`, `instoptembnd`

optembndbyhjm

Purpose

Price bonds with embedded options by Heath-Jarrow-Morton interest-rate tree

Syntax

```
[Price, PriceTree] = optembndbyhjm(HJMTree, CouponRate,
Settle, Maturity, OptSpec, Strike, ExerciseDates,
'Name1',  Value1, 'Name2', Value2, ...)
```

Arguments

`HJMTree`	Interest-rate tree structure created by `hjmtree`.
`CouponRate`	`NINST`-by-1 matrix for the decimal annual rate.
`Settle`	`NINST`-by-1 matrix for the settlement date. A vector of serial date numbers or date strings. `Settle` must be earlier than `Maturity`.
`Maturity`	`NINST`-by-1 matrix for the maturity date. A vector of serial date numbers or date strings.
`OptSpec`	`NINST`-by-1 cell array of strings `'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 `NaN`s. For an American option: `NINST`-by-1 vector of strike price values for each option.

`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.
`'Name1', Value1, 'Name2', Value2 ...`	(Optional) The name/value pairs are a variable length list of parameters. All optional inputs are specified as matching parameter name/value pairs. The parameter name is specified as a character string, followed by the corresponding parameter value. Parameter name/value pairs may be specified in any order; names are case insensitive and partial string matches are allowed provided no ambiguities exist. Valid parameter names are as follows: • `AmericanOpt` is a `NINST`-by-`1` matrix for flags options: `0` (European/Bermuda) or `1` (American). Default is `0`. • `Period` is a `NINST`-by-`1` matrix for coupons per year. Default is `2`.

- `Basis` is a day-count basis of the instrument. `Basis` is a vector of integers with the following supported values:
  - 0 = actual/actual (default)
  - 1 = 30/360 (SIA)
  - 2 = actual/360
  - 3 = actual/365
  - 4 = 30/360 (BMA)
  - 5 = 30/360 (ISDA)
  - 6 = 30/360 (European)
  - 7 = actual/365 (Japanese)
  - 8 = actual/actual (ICMA)
  - 9 = actual/360 (ICMA)
  - 10 = actual/365 (ICMA)
  - 11 = 30/360E (ICMA)
  - 12 = actual/actual (ISDA)
  - 13 = BUS/252
- `EndMonthRule` is a `NINST`-by-`1` matrix for the end-of-month rule. This rule applies only when `Maturity` is an end-of-month date for a month having 30 or fewer days. When the value is `0`, the end-of-month rule is ignored, meaning that a bond's coupon payment date is always the same numerical day of the month. When the value is `1`, the end-of-month rule is set on (default), meaning that a bond's coupon payment date is always the last actual day of the month.

- `IssueDate` is a `NINST`-by-`1` matrix for the bond issue date.
- `FirstCouponDate` is a `NINST`-by-`1` matrix for an irregular first coupon date. 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` is a `NINST`-by-`1` matrix for an irregular last coupon date. Last coupon date of a bond before the maturity date. In the absence of a specified `FirstCouponDate`, a specified `LastCouponDate` determines the coupon structure of the bond. The coupon structure of a bond is truncated at the `LastCouponDate` regardless of where it falls and will be followed only by the bond's maturity cash flow date.
- `StartDate` is a `NINST`-by-`1` matrix for date when a bond actually starts (the date from which a bond cash flow is considered). To make an instrument forward-starting, specify this date as a future date. If you do not specify `StartDate`, the effective start date is the `Settle` date.
- `Face` is a `NINST`-by-`1` matrix for the face value. The default value is `100`.
- `Options` is a derivatives pricing options structure created with `derivset`.

Note The `Settle` date for every bond with embedded option is set to the `ValuationDate` of the HJM tree; the bond's argument for `Settle` date is ignored.

Description

`[Price, PriceTree] = optembndbyhjm(HJMTree, CouponRate,Settle, Maturity, OptSpec, Strike, ExerciseDates,'Name1', Value1, 'Name2', Value2, ...)` prices bonds with embedded options by an HJM 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.PBush` contains the clean prices.
- `PriceTree.AIBush` contains the accrued interest.
- `PriceTree.tObs` contains the observation times.

Examples

Create an `HJMTree` with the following data:

```
Rates = [0.05;0.06;0.07];
Compounding = 1;
StartDates = ['jan-1-2007';'jan-1-2008';'jan-1-2009'];
EndDates   = ['jan-1-2008';'jan-1-2009';'jan-1-2010'];
ValuationDate = 'jan-1-2007';
```

Create a `RateSpec`:

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

Specify a `VolSpec`:

```
VolSpec = hjmvolspec('Constant', 0.01);
```

Specify a `TimeSpec`:

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

Build an `HJMTree`:

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

To compute the price of an American callable bond that pays a 6% annual coupon and matures and is callable on January 1, 2010:

```
BondSettlement = 'jan-1-2007';
BondMaturity   = 'jan-1-2010';
CouponRate = 0.06;
Period = 1;
OptSpec = 'call';
Strike = [98];
ExerciseDates = '01-Jan-2010';
AmericanOpt = 1;

[PriceCallBond,PT] = optembndbyhjm(HJMTree, CouponRate, BondSettlement, BondMaturity,...
OptSpec, Strike, ExerciseDates, 'Period', 1,'AmericanOp',1)

PriceCallBond =

95.9492

PT =

FinObj: 'HJMPriceTree'
      tObs: [0 1 2 3]
     PBush: {[95.9492]  [1x1x2 double]  [1x2x2 double]  [98 98 98 98]}
```

See Also

`hjmprice`, `hjmtree`, `instoptembnd`

optembndbyhw

Purpose

Price bonds with embedded options by Hull-White interest-rate tree

Syntax

```
[Price, PriceTree] = optembndbyhw(HWTree, CouponRate,
Settle, Maturity, OptSpec, Strike, ExerciseDates,
'Name1', Value1, 'Name2', Value2, ...)
```

Arguments

`HWTree`	Interest-rate tree structure created by `hwtree`.
`CouponRate`	`NINST`-by-1 matrix for the decimal annual rate.
`Settle`	`NINST`-by-1 matrix for the settlement date. A vector of serial date numbers or date strings. `Settle` must be earlier than `Maturity`.
`Maturity`	`NINST`-by-1 matrix for the maturity date. A vector of serial date numbers or date strings.
`OptSpec`	`NINST`-by-1 cell array of strings `'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 `NaN`s. For an American option: `NINST`-by-1 vector of strike price values for each option.

`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.

`'Name1', Value1`
`'Name2', Value2`
`...`
	(Optional) The name/value pairs are a variable length list of parameters. All optional inputs are specified as matching parameter name/value pairs. The parameter name is specified as a character string, followed by the corresponding parameter value. Parameter name/value pairs may be specified in any order; names are case insensitive and partial string matches are allowed provided no ambiguities exist. Valid parameter names are as follows:
	
	• `AmericanOpt` is a `NINST`-by-`1` matrix for flags options: `0` (European/Bermuda) or `1` (American). Default is `0`.
	
	• `Period` is a `NINST`-by-`1` matrix for coupons per year. Default is `2`.

- `Basis` is a day-count basis of the instrument. `Basis` is a vector of integers with the following supported values:
  - 0 = actual/actual (default)
  - 1 = 30/360 (SIA)
  - 2 = actual/360
  - 3 = actual/365
  - 4 = 30/360 (BMA)
  - 5 = 30/360 (ISDA)
  - 6 = 30/360 (European)
  - 7 = actual/365 (Japanese)
  - 8 = actual/actual (ICMA)
  - 9 = actual/360 (ICMA)
  - 10 = actual/365 (ICMA)
  - 11 = 30/360E (ICMA)
  - 12 = actual/actual (ISDA)
  - 13 = BUS/252
- `EndMonthRule` is a `NINST`-by-`1` matrix for the end-of-month rule. This rule applies only when `Maturity` is an end-of-month date for a month having 30 or fewer days. When the value is `0`, the end-of-month rule is ignored, meaning that a bond's coupon payment date is always the same numerical day of the month. When the value is `1`, the end-of-month rule is set rule on (default), meaning that a bond's coupon payment date is always the last actual day of the month.

- `IssueDate` is a `NINST`-by-`1` matrix for the bond issue date.
- `FirstCouponDate` is a `NINST`-by-`1` matrix for an irregular first coupon date. 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` is a `NINST`-by-`1` matrix for an irregular last coupon date. Last coupon date of a bond before the maturity date. In the absence of a specified `FirstCouponDate`, a specified `LastCouponDate` determines the coupon structure of the bond. The coupon structure of a bond is truncated at the `LastCouponDate` regardless of where it falls and will be followed only by the bond's maturity cash flow date.
- `StartDate` is a `NINST`-by-`1` matrix for date when a bond actually starts (the date from which a bond cash flow is considered). To make an instrument forward-starting, specify this date as a future date. If you do not specify `StartDate`, the effective start date is the `Settle` date.
- `Face` is a `NINST`-by-`1` matrix for the face value. The default value is `100`.
- `Options` is a derivatives pricing options structure created with `derivset`.

Note The Settle date for every bond with embedded option is set to the ValuationDate of the HW tree; the bond's argument for Settle date is ignored.

Description

[Price, PriceTree] = optembndbyhw(HWTree, CouponRate, Settle, Maturity, OptSpec, Strike, ExerciseDates,'Name1', Value1, 'Name2', Value2, ...) prices bonds with embedded options by a HW 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

Create a HWTree with the following data:

```
ZeroRates = [ 0.035;0.04;0.045];
Compounding = 1;
StartDates = ['jan-1-2007';'jan-1-2008';'jan-1-2009'];
EndDates   = ['jan-1-2008';'jan-1-2009';'jan-1-2010'];
ValuationDate = 'jan-1-2007';
```

Create a RateSpec:

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

Specify a TimeSpec:

```
HWTimeSpec = hwtimespec(ValuationDate, EndDates, Compounding);
```

Specify a `VolSpec`:

```
VolDates = ['jan-1-2008';'jan-1-2009';'jan-1-2010'];
VolCurve = 0.01;
AlphaDates = 'jan-1-2010';
AlphaCurve = 0.1;
HWVolSpec = hwvolspec(ValuationDate, VolDates, VolCurve, AlphaDates, AlphaCurve);
```

Build a `HWTree`:

```
HWTree = hwtree(HWVolSpec, RateSpec, HWTimeSpec);
```

Compute the price of an American puttable bond that pays an annual coupon of 5.25%, matures on January 1, 2010, and is puttable from January 1, 2008 to January 1, 2010:

```
BondSettlement = 'jan-1-2007';
BondMaturity   = 'jan-1-2010';
CouponRate = 0.0525;
Period = 1;
OptSpec = 'put';
Strike = [100];
ExerciseDates = {'jan-1-2008' '01-Jan-2010'};
AmericanOpt = 1;

PricePutBondHW = optembndbyhw(HWTree, CouponRate, BondSettlement, BondMaturity,...
OptSpec, Strike, ExerciseDates,'Period', 1, 'AmericanOpt', 1)

PricePutBondHW =

102.8801
```

See Also

`hwprice`, `hwtree`, `instoptembnd`

optstockbybjs

Purpose

Price American options using Bjerksund-Stensland 2002 option pricing model

Syntax

```
Price = optstockbybjs(RateSpec, StockSpec, Settle, Maturity,
OptSpec, Strike)
```

Arguments

`RateSpec`	The annualized continuously compounded rate term structure. For information on the interest rate specification, see `intenvset`.
`StockSpec`	Stock specification. See `stockspec`.
`Settle`	`NINST`-by-1 vector of settlement or trade dates.
`Maturity`	`NINST`-by-1 vector of maturity dates.
`OptSpec`	`NINST`-by-1 cell array of strings `'call'` or `'put'`.
`Strike`	`NINST`-by-1 vector of strike price values.

Description

`Price = optstockbybjs(RateSpec, StockSpec, Settle, Maturity, OptSpec, Strike)` computes American option prices with continuous dividend yield using the Bjerksund-Stensland 2002 option pricing model.

`Price` is a `NINST`-by-1 vector of expected option prices.

Examples

Consider two American stock options (a call and a put) with an exercise price of $100. The options expire on April 1, 2008. Assume the underlying stock pays a continuous dividend yield of 4% as of January 1, 2008. The stock has a volatility of 20% per annum and the annualized continuously compounded risk-free rate is 8% per annum. Using this data, calculate the price of the American call and put, assuming the following current prices of the stock: $90 (for the call) and $120 (for the put):

```
Settle = 'Jan-1-2008';
Maturity = 'April-1-2008';
Strike = 100;
AssetPrice = [90;120];
DivYield = 0.04;
Rate = 0.08;
Sigma = 0.20;
```

Define `StockSpec` and `RateSpec`:

```
StockSpec = stockspec(Sigma, AssetPrice, {'continuous'}, DivYield);

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

Define the option type:

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

Compute the option prices using the Bjerksund-Stensland 2002 option pricing model:

```
Price = optstockbybjs(RateSpec, StockSpec, Settle, Maturity, OptSpec, Strike)

Price =

    0.8420
    0.1108
```

The first element of the `Price` vector represents the price of the call ($0.84); the second element represents the price of the put option ($0.11).

See Also

`impvbybjs`, `intenvset`, `optstocksensbybjs`, `stockspec`

optstockbyblk

Purpose

Price options on futures using Black option pricing model

Syntax

```
Price = optstockbyblk(RateSpec, StockSpec, Settle, Maturity,
OptSpec, Strike)
```

Arguments

`RateSpec`	The annualized continuously compounded rate term structure. For information on the interest rate specification, see `intenvset`.
`StockSpec`	Stock specification. See `stockspec`.
`Settle`	`NINST`-by-1 vector of settlement or trade dates.
`Maturity`	`NINST`-by-1 vector of maturity dates.
`OptSpec`	`NINST`-by-1 cell array of strings `'call'` or `'put'`.
`Strike`	`NINST`-by-1 vector of strike price values.

Description

`Price = optstockbyblk(RateSpec, StockSpec, Settle, Maturity, OptSpec, Strike)` computes option prices on futures using the Black option pricing model.

`Price` is a `NINST`-by-1 vector of expected option prices.

Examples

Consider two European call options on a futures contract with exercise prices of $20 and $25 that expire on September 1, 2008. Assume that on May 1, 2008 the contract is trading at $20, and has a volatility of 35% per annum. The risk-free rate is 4% per annum. Using this data, calculate the price of the call futures options using the Black model:

```
Strike = [20; 25];
AssetPrice = 20;
Sigma = .35;
Rates = 0.04;
Settle = 'May-01-08';
```

```
Maturity = 'Sep-01-08';
```

Create `RateSpec` and `StockSpec`:

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

StockSpec = stockspec(Sigma, AssetPrice);
```

Define the call options:

```
OptSpec = {'call'};
```

Calculate the price using the Black option pricing model:

```
Price = optstockbyblk(RateSpec, StockSpec, Settle, Maturity,...
OptSpec, Strike)
Price =
    1.5903
    0.3037
```

See Also

`impvbyblk`, `intenvset`, `optstocksensbyblk`, `stockspec`

optstockbybls

Purpose

Price options using Black-Scholes option pricing model

Syntax

```
Price = optstockbybls(RateSpec, StockSpec, Settle, Maturity,
OptSpec, Strike)
```

Arguments

`RateSpec`	The annualized continuously compounded rate term structure. For information on the interest rate specification, see `intenvset`.
`StockSpec`	Stock specification. See `stockspec`.
`Settle`	`NINST`-by-1 vector of settlement or trade dates.
`Maturity`	`NINST`-by-1 vector of maturity dates.
`OptSpec`	`NINST`-by-1 cell array of strings `'call'` or `'put'`.
`Strike`	`NINST`-by-1 vector of strike price values.

Description

`Price = optstockbybls(RateSpec, StockSpec, Settle, Maturity, OptSpec, Strike)` computes option prices using the Black-Scholes option pricing model.

`Price` is a `NINST`-by-1 vector of expected option prices.

Note When using `StockSpec` with `optstockbybls`, you can modify `StockSpec` to handle other types of underliers when pricing instruments that use the Black-Scholes model.

When pricing Futures (Black model), enter the following in `StockSpec`:

```
DivType = 'Continuous';
DivAmount = RateSpec.Rates;
```

When pricing Foreign Currencies (Garman-Kohlhagen model), enter the following in `StockSpec`:

```
DivType = 'Continuous';
DivAmount = ForeignRate;
```

where `ForeignRate` is the continuously compounded, annualized risk free interest rate in the foreign country.

Examples

Consider two European options, a call and a put, with an exercise price of $29 on June 1, 2008. The options expire on May 1, 2008. Assume that the underlying stock for the call option provides a cash dividend of $0.50 on February 15, 2008. The underlying stock for the put option provides a continuous dividend yield of 4.5% per annum. The stocks are trading at $30 and have a volatility of 25% per annum. The annualized continuously compounded risk-free rate is 5% per annum. Using this data, compute the price of the options using the Black-Scholes model:

```
Strike = 29;
AssetPrice = 30;
Sigma = .25;
Rates = 0.05;
Settle = 'Jan-01-2008';
Maturity = 'May-01-2008';
```

Define `RateSpec`:

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

Define `StockSpec`:

```
DividendType = {'cash';'continuous'};
DividendAmounts = [0.50; 0.045];
ExDividendDates = {'Feb-15-2008';NaN};

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

Price the call and the put options using the Black-Scholes model:

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

Price = optstockbybls(RateSpec, StockSpec, Settle, Maturity, OptSpec, Strike)

Price =

    2.2030
    1.2025
```

See Also

`impvbybls`, `intenvset`, `optstocksensbybls`, `stockspec`

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 `NaN`s. 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

optstockbyeqp

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`	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 `NaN`s. 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] = 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

Purpose

Price options on stocks using implied trinomial tree (ITT)

Syntax

```
[Price, PriceTree] = optstockbyitt(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'`.

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 `NaN`s. American option: `NINST`-by-`1` vector of strike price values for each option.
`Settle`	`NINST`-by-`1` vector of settlement or trade dates.

`ExerciseDates`	For a European or Bermuda option: `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.
`AmericanOpt`	(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.

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(ITTTree, 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 `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.

```
OptSpec = 'Put';
Strike = 80;
Settle = '01-Jan-2006';
ExerciseDates = ' 01-Jan-2010 ';


Price = optstockbyitt(ITTTree, OptSpec, Strike, Settle,ExerciseDates)

Price =

 10.68
```

See Also

`instoptstock`, `itttree`, `stockoptspec`

optstockbyrgw

Purpose

Calculate American call option prices using Roll-Geske-Whaley option pricing model

Syntax

```
Price = optstockbyrgw(RateSpec, StockSpec, Settle,
Maturity, Strike)
```

Arguments

`RateSpec`	The annualized continuously compounded rate term structure. For information on the interest rate specification, see `intenvset`.
`StockSpec`	Stock specification. See `stockspec`.
`Settle`	`NINST`-by-`1` vector of settlement or trade dates.
`Maturity`	`NINST`-by-`1` vector of maturity dates.
`Strike`	`NINST`-by-`1` vector of strike price values.

Description

`Price = optstockbyrgw(RateSpec, StockSpec, Settle, Maturity, Strike)` computes the American call option prices using the Roll-Geske-Whaley option pricing model.

`Price` is a `NINST`-by-1 vector of expected call option prices.

Note `optstockbyrgw` computes prices of American calls with a single cash dividend using the Roll-Geske-Whaley option pricing model.

Examples

Consider an American call option with an exercise price of $22 that expires on February 1, 2009. The underlying stock is trading at $20 on June 1, 2008 and has a volatility of 20% per annum. The annualized continuously compounded risk-free rate is 6.77% per annum. The stock pays a single dividend of $2 on September 1, 2008. Using this data,

compute price of the American call option using the Roll-Geske-Whaley option pricing model:

```
Settle = 'Jun-01-2008';
Maturity = 'Feb-01-2009';
AssetPrice = 20;
Strike = 22;
Sigma  = 0.2;
Rate = 0.0677;
```

Define `StockSpec` and `RateSpec`:

```
RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle, 'EndDates',...
Maturity, 'Rates', Rate, 'Compounding', -1, 'Basis', 0);

StockSpec = stockspec(Sigma, AssetPrice, {'cash'}, DivAmount, DivDate);
```

Compute the price of the American call :

```
Price  = optstockbyrgw(RateSpec, StockSpec, Settle, Maturity,Strike)

Price =

    0.3359
```

See Also

`impvbyrgw`, `intenvset`, `optstocksensbyrgw`, `stockspec`

optstocksensbybjs

Purpose

Calculate American option prices and sensitivities using Bjerksund-Stensland 2002 option pricing model

Syntax

```
PriceSens = optstocksensbybjs(RateSpec, StockSpec, Settle,
Maturity, OptSpec, Strike, 'Name1', Value1...)
```

Arguments

`RateSpec`	The annualized continuously compounded rate term structure. For information on the interest rate specification, see `intenvset`.
`StockSpec`	Stock specification. See `stockspec`.
`Settle`	`NINST`-by-`1` vector of settlement or trade dates.
`Maturity`	`NINST`-by-`1` vector of maturity dates.
`OptSpec`	`NINST`-by-`1` cell array of strings `'call'` or `'put'`.
`Strike`	`NINST`-by-`1` vector of strike price values.
`OutSpec`	(Optional) All optional inputs are specified as matching parameter name/value pairs. The parameter name is specified as a character string, followed by the corresponding parameter value. Parameter name/value pairs may be specified in any order; names are case-insensitive and partial string matches are allowed provided no ambiguities exist. Valid parameter names are: • `NOUT`-by-`1` or `1`-by-`NOUT` cell array of strings indicating the nature and order of the outputs for the function. Possible values are: `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, or `'All'`. For example, `OutSpec = {'Price'; 'Lamba'; 'Rho'}` specifies that the output should be `Price`, `Lambda`, and `Rho`, in that order.

To invoke from a function: `[Price, Lambda, Rho] = optstocksensbybjs(..., 'OutSpec', {'Price', 'Lamba', 'Rho'})`

`OutSpec = {'All'}` specifies that the output should be `Delta`, `Gamma`, `Vega`, `Lambda`, `Rho`, `Theta`, and `Price`, in that order. This is the same as specifying `OutSpec` as `OutSpec = {'Delta', 'Gamma', 'Vega', 'Lambda', 'Rho', 'Theta', 'Price'};`.

- Default is `OutSpec = {'Price'}`.

Description

`PriceSens = optstocksensbybjs(RateSpec, StockSpec, Settle, Maturity, OptSpec, Strike, 'Name1', Value1...)` computes American option prices and sensitivities using the Bjerksund-Stensland 2002 option pricing model.

`optstocksensbybjs` can be used to compute six sensitivities for the Bjerksund-Stensland 2002 model: `delta`, `gamma`, `vega`, `lambda`, `rho`, and `theta`. This function is also capable of returning the price of the option. The selection of output parameters and their order is determined by the optional input parameter `OutSpec`. This parameter is a cell array of strings, each one specifying a desired output parameter. The order in which these output parameters are returned by the function is the same as the order of the strings contained in `OutSpec`.

`PriceSens` is a `NINST`-by-`1` vector of expected prices or sensitivities values.

Examples

Consider four American put options with an exercise price of $100. The options expire on October 1, 2008. Assume the underlying stock pays a continuous dividend yield of 4% and has a volatility of 40% per annum. The annualized continuously compounded risk-free rate is 8% per annum. Using this data, calculate the `delta`, `gamma`, and `price` of the American put options, assuming the following current prices of the stock on July 1, 2008: $90, $100, $110 and $120:

```
Settle = 'July-1-2008';
Maturity = 'October-1-2008';
Strike = 100;
AssetPrice = [90;100;110;120];
Rate = 0.08;
Sigma = 0.40;
DivYield = 0.04;
```

Define `StockSpec` and `RateSpec`:

```
StockSpec = stockspec(Sigma, AssetPrice, {'continuous'}, DivYield);

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

Define the option type:

```
OptSpec = {'put'};
```

Compute `delta`, `gamma`, and `price` of the put options using the Bjerksund-Stensland 2002 option pricing model:

```
OutSpec = {'Delta', 'Gamma', 'Price'};

[Delta, Gamma, Price] = optstocksensbybjs(RateSpec, StockSpec, Settle, Maturity,...
OptSpec, Strike, 'OutSpec', OutSpec)

Delta =

    0.7210
    0.5179
    0.3321
    0.1925


Gamma =
```

```
    0.0176
    0.0202
    0.0182
    0.0136


Price =

   12.9467
    7.4571
    3.9539
    1.9495
```

See Also

impvbybjs, intenvset, optstockbybjs, stockspec

optstocksensbyblk

Purpose

Calculate option prices and sensitivities on futures using Black pricing model

Syntax

```
PriceSens = optstocksensbyblk(RateSpec, StockSpec, Settle,
Maturity, OptSpec, Strike, 'Name1', Value1...)
```

Arguments

`RateSpec`	The annualized continuously compounded rate term structure. For information on the interest rate specification, see `intenvset`.
`StockSpec`	Stock specification. See `stockspec`.
`Settle`	`NINST`-by-`1` vector of settlement or trade dates.
`Maturity`	`NINST`-by-`1` vector of maturity dates.
`OptSpec`	`NINST`-by-`1` cell array of strings `'call'` or `'put'`.
`Strike`	`NINST`-by-`1` vector of strike price values.
`OutSpec`	(Optional) All optional inputs are specified as matching parameter name/value pairs. The parameter name is specified as a character string, followed by the corresponding parameter value. Parameter name/value pairs may be specified in any order; names are case-insensitive and partial string matches are allowed provided no ambiguities exist. Valid parameter names are: • `NOUT`-by-`1` or `1`-by-`NOUT` cell array of strings indicating the nature and order of the outputs for the function. Possible values are: `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, or `'All'`. For example, `OutSpec = {'Price'; 'Lamba'; 'Rho'}` specifies that the output should be `Price`, `Lambda`, and `Rho`, in that order.

To invoke from a function: `[Price, Lambda, Rho] = optstocksensbyblk(..., 'OutSpec', {'Price', 'Lamba', 'Rho'})`

`OutSpec = {'All'}` specifies that the output should be `Delta`, `Gamma`, `Vega`, `Lambda`, `Rho`, `Theta`, and `Price`, in that order. This is the same as specifying `OutSpec` as `OutSpec = {'Delta', 'Gamma', 'Vega', 'Lambda', 'Rho', 'Theta', 'Price'};`.

- Default is `OutSpec = {'Price'}`.

Description

`PriceSens = optstocksensbyblk(RateSpec, StockSpec, Settle, Maturity, OptSpec, Strike, 'Name1', Value1...)` computes option prices and sensitivities on futures using the Black pricing model.

`PriceSens` is a `NINST`-by-1 vector of expected future prices or sensitivities values.

Examples

Consider a European put option on a futures contract with an exercise price of $60 that expires on June 30, 2008. On April 1, 2008 the underlying stock is trading at $58 and has a volatility of 9.5% per annum. The annualized continuously compounded risk-free rate is 5% per annum. Using this data, compute `delta`, `gamma`, and the `price` of the put option.

```
AssetPrice = 58;
Strike = 60;
Sigma = .095;
Rates = 0.05;
Settle = 'April-01-08';
Maturity = 'June-30-08';
```

Create `RateSpec` and `StockSpec`:

```
RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle, 'EndDates',...
```

```
Maturity, 'Rates', Rates, 'Compounding', -1, 'Basis', 1);

StockSpec = stockspec(Sigma, AssetPrice);
```

Define the options:

```
OptSpec = {'put'};
```

Compute `Delta`, `Gamma` and `Price` for the European put option:

```
OutSpec = {'Delta','Gamma','Price'};
[Delta, Gamma, Price] = optstocksensbyblk(RateSpec, StockSpec, Settle,...
Maturity, OptSpec, Strike,'OutSpec', OutSpec)

Delta =

   -0.7469


Gamma =

    0.1130


Price =

    2.3569
```

See Also

`impvbyblk`, `intenvset`, `optstockbyblk`, `stockspec`

Purpose

Calculate option prices and sensitivities using Black-Scholes option pricing model

Syntax

```
PriceSens = optstocksensbybls(RateSpec, StockSpec, Settle,
Maturity, OptSpec, Strike, 'Name1', Value1...)
```

Arguments

`RateSpec`	The annualized continuously compounded rate term structure. For information on the interest rate specification, see `intenvset`.
`StockSpec`	Stock specification. See `stockspec`.
`Settle`	`NINST`-by-`1` vector of settlement or trade dates.
`Maturity`	`NINST`-by-`1` vector of maturity dates.
`OptSpec`	`NINST`-by-`1` cell array of strings `'call'` or `'put'`.
`Strike`	`NINST`-by-`1` vector of strike price values.
`OutSpec`	(Optional) All optional inputs are specified as matching parameter name/value pairs. The parameter name is specified as a character string, followed by the corresponding parameter value. Parameter name/value pairs may be specified in any order; names are case-insensitive and partial string matches are allowed provided no ambiguities exist. Valid parameter names are: • `NOUT`-by-`1` or `1`-by-`NOUT` cell array of strings indicating the nature and order of the outputs for the function. Possible values are: `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, or `'All'`. For example, `OutSpec = {'Price'; 'Lamba'; 'Rho'}` specifies that the output should be `Price`, `Lambda`, and `Rho`, in that order.

To invoke from a function: `[Price, Lambda, Rho] = optstocksensbybls(..., 'OutSpec', {'Price', 'Lamba', 'Rho'})`

`OutSpec = {'All'}` specifies that the output should be `Delta`, `Gamma`, `Vega`, `Lambda`, `Rho`, `Theta`, and `Price`, in that order. This is the same as specifying `OutSpec` as `OutSpec = {'Delta', 'Gamma', 'Vega', 'Lambda', 'Rho', 'Theta', 'Price'};`.

- Default is `OutSpec = {'Price'}`.

Description

`PriceSens = optstocksensbybls(RateSpec, StockSpec, Settle, Maturity, OptSpec, Strike, 'Name1', Value1...)` computes option prices and sensitivities using the Black-Scholes option pricing model.

`PriceSens` is a `NINST`-by-`1` vector of expected prices or sensitivities values.

Note When using `StockSpec` with `optstocksensbybls`, you can modify `StockSpec` to handle other types of underliers when pricing instruments that use the Black-Scholes model.

When pricing Futures (Black model), enter the following in `StockSpec`:

```
DivType = 'Continuous';
DivAmount = RateSpec.Rates;
```

When pricing Foreign Currencies (Garman-Kohlhagen model), enter the following in `StockSpec`:

```
DivType = 'Continuous';
DivAmount = ForeignRate;
```

where `ForeignRate` is the continuously compounded, annualized risk free interest rate in the foreign country.

Examples

Consider a European call and put options with an exercise price of $30 that expires on June 1, 2008. The underlying stock is trading at $30 on January 1, 2008 and has a volatility of 30% per annum. The annualized continuously compounded risk-free rate is 5% per annum. Using this data, compute the `delta`, `gamma`, and `price` of the options using the Black-Scholes model.

```
AssetPrice = 30;
Strike = 30;
Sigma = .30;
Rates = 0.05;
Settle = 'January-01-2008';
Maturity = 'June -01-2008';
```

Define `RateSpec` and `StockSpec` :

```
RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle, 'EndDates',...
```

```
Maturity, 'Rates', Rates, 'Compounding',-1, 'Basis', 1);

StockSpec = stockspec(Sigma, AssetPrice);
```

Define the options:

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

Compute `delta`, `gamma`, and `price` for the European options:

```
OutSpec = {'Delta','Gamma','Price'};
[Delta, Gamma, Price] = optstocksensbybls(RateSpec, StockSpec, Settle,...
Maturity, OptSpec, Strike,'OutSpec', OutSpec)

Delta =

    0.5810
   -0.4190


Gamma =

    0.0673
    0.0673


Price =

    2.6126
    1.9941
```

See Also

`impvbybls`, `intenvset`, `optstockbybls`, `stockspec`

Purpose

Calculate American call option prices and sensitivities using Roll-Geske-Whaley option pricing model

Syntax

```
PriceSens = optstocksensbyrgw(RateSpec, StockSpec, Settle,
Maturity, OptSpec, Strike, 'Name1', Value1...)
```

Arguments

`RateSpec`	The annualized continuously compounded rate term structure. For information on the interest rate specification, see `intenvset`.
`StockSpec`	Stock specification. See `stockspec`.
`Settle`	`NINST`-by-`1` vector of settlement or trade dates.
`Maturity`	`NINST`-by-`1` vector of maturity dates.
`OptSpec`	`NINST`-by-`1` cell array of strings `'call'` or `'put'`.
`Strike`	`NINST`-by-`1` vector of strike price values.
`OutSpec`	(Optional) All optional inputs are specified as matching parameter name/value pairs. The parameter name is specified as a character string, followed by the corresponding parameter value. Parameter name/value pairs may be specified in any order; names are case-insensitive and partial string matches are allowed provided no ambiguities exist. Valid parameter names are: • `NOUT`-by-`1` or `1`-by-`NOUT` cell array of strings indicating the nature and order of the outputs for the function. Possible values are: `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, or `'All'`. For example, `OutSpec = {'Price'; 'Lamba'; 'Rho'}` specifies that the output should be `Price`, `Lambda`, and `Rho`, in that order.

To invoke from a function: `[Price, Lambda, Rho] = optstocksensbyrgw(..., 'OutSpec', {'Price', 'Lamba', 'Rho'})`

`OutSpec = {'All'}` specifies that the output should be `Delta`, `Gamma`, `Vega`, `Lambda`, `Rho`, `Theta`, and `Price`, in that order. This is the same as specifying `OutSpec` as `OutSpec = {'Delta', 'Gamma', 'Vega', 'Lambda', 'Rho', 'Theta', 'Price'};`.

- Default is `OutSpec = {'Price'}`.

Description

`PriceSens = optstocksensbyrgw(RateSpec, StockSpec, Settle, Maturity, OptSpec, Strike, 'Name1', Value1...)` computes American call option prices and sensitivities using the Roll-Geske-Whaley option pricing model.

`PriceSens` is a `NINST`-by-`1` vector of expected prices or sensitivities values.

Examples

Consider an American stock option with an exercise price of $82 on January 1, 2008 that expires on May 1, 2008. Assume the underlying stock pays dividends of $4 on April 1, 2008. The stock is trading at $80 and has a volatility of 30% per annum. The risk-free rate is 6% per annum. Using this data, calculate the price and the value of `delta` and `gamma` of the American call using the Roll-Geske-Whaley option pricing model:

```
AssetPrice = 80;
Settle = 'Jan-01-2008';
Maturity = 'May-01-2008';
Strike = 82;
Rate = 0.06;
Sigma  = 0.3;
DivAmount = 4;
```

```
DivDate = 'Apr-01-2008';
```

Define `StockSpec` and `RateSpec`:

```
StockSpec = stockspec(Sigma, AssetPrice, {'cash'}, DivAmount, DivDate);

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

Define `OutSpec`:

```
OutSpec = {'Price', 'Delta', 'Gamma'};
```

Calculate the call `Price`, `Delta`, and `Gamma`:

```
[Price, Delta, Gamma]  = optstocksensbyrgw(RateSpec, StockSpec, Settle,...
Maturity, Strike,'OutSpec', OutSpec)

Price =

    4.3860


Delta =

    0.5022


Gamma =

    0.0336
```

See Also

`impvbyrgw`, `intenvset`, `optstockbyrgw`, `stockspec`

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 1 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. `StartDates` must be earlier than `EndDates`. 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. • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ICMA)

- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

`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 = Tsemi/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 1-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`

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 1 year.

`Compounding` = 365

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

`Compounding` = -1

	`Disc = exp(-T*Z)`, where `T` is time in years.
`RefRates`	`NREFPTS`-by-`NCURVES` matrix of reference rates in decimal form. `RefRates` are the yields over investment intervals from `RefStartTimes`, when the cash flow is valued, to `RefEndTimes`, when the cash flow is received.
`RefEndTimes`	`NREFPTS`-by-`1` vector or scalar of times in periodic units ending the intervals corresponding to `RefRates`.
`RefStartTimes`	(Optional) `NREFPTS`-by-`1` vector or scalar of times in periodic units starting the intervals corresponding to `RefRates`. Default = `0`.
`EndTimes`	`NPOINTS`-by-`1` vector or scalar of times in periodic units ending the interval to discount over.
`StartTimes`	(Optional) `NPOINTS`-by-`1` vector or scalar of times in periodic units starting the interval to discount over. Default = `0`.
`RefEndDates`	`NREFPTS`-by-`1` vector or scalar of serial dates ending the intervals corresponding to `RefRates`.
`RefStartDates`	(Optional) `NREFPTS`-by-`1` vector or scalar of serial dates starting the intervals corresponding to `RefRates`. Default = `ValuationDate`.
`EndDates`	`NPOINTS`-by-`1` vector or scalar of serial maturity dates ending the interval to discount over.

StartDates	(Optional) NPOINTS-by-1 vector or scalar of serial dates starting the interval to discount over. StartDates must be earlier than EndDates. 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 piece-wise linear in time.

`Rates` is an `NPOINTS`-by-`NCURVES` matrix of rates implied by the reference interest-rate structure and sampled at new intervals.

`StartTimes` is an `NPOINTS`-by-`1` column vector of times starting the new intervals where rates are desired, measured in periodic units.

`EndTimes` is an `NPOINTS`-by-`1` column vector of times ending the new intervals, measured in periodic units.

If `Compounding = 365` (daily), `StartTimes` and `EndTimes` are measured in days. The arguments otherwise contain values, `T`, computed from SIA semiannual time factors, `Tsemi`, by the formula `T = Tsemi/2 * F`, where `F` is the compounding frequency.

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 1-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`

Purpose

Specify European stock option structure

Syntax

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

Arguments

`OptPrice`	`NINST`-by-1 vector of European option prices.
`Strike`	`NINST`-by-1 vector of strike prices.
`Settle`	Scalar date marking the settlement date.
`Maturity`	`NINST`-by-1 vector of maturity dates.
`OptSpec`	`NINST`-by-1 cell array of strings `'call'` or `'put'`.
`InterpMethod`	(Optional) Method of interpolation to use for option prices. `InterpMethod` is `[{'price'} | 'vol']`. 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';
    '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'
```

See Also

ittprice, itttree, stockspec

Purpose

Create stock structure

Syntax

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

Arguments

`Sigma`	`NINST`-by-`1` decimal annual price volatility of underlying security.
`AssetPrice`	`NINST`-by-`1` vector of underlying asset price values at time 0.
`DividendType`	(Optional) `NINST`-by-`1` cell array of strings specifying each stock's dividend type. Dividend type must be either `cash` for actual dollar dividends, `constant` for constant dividend yield, or `continuous` for continuous dividend yield. This function does not handle stock option dividends.

Note Dividends are assumed to be paid in cash. Noncash dividends (stock) are not allowed. When combining two or more type of dividends, shorter rows should be padded with the value `NaN`.

`DividendAmounts`	(Optional) `NINST`-by-`NDIV` matrix of cash dividends or `NINST`-by-`1` vector representing a constant or continuous annualized dividend yield.
`ExDividendDates`	(Optional) `NINST`-by-`NDIV` matrix of ex-dividend dates for `cash` type or `NINST`-by-`1` vector of ex-dividend dates for `constant` dividend type. For `continuous` dividend type, this argument should be ignored.

Description

`StockSpec = stockspec(Sigma, AssetPrice, DividendType, DividendAmounts, ExDividendDates)` creates a MATLAB structure containing the properties of a stock.

Examples

Example 1. Consider a stock that provides four cash dividends of $0.50 on January 3, 2008, April 1, 2008, July 5, 2008 and October 1, 2008. The stock is trading at $50, and has a volatility of 20% per annum. Using this data, create the structure `StockSpec`:

```
AssetPrice = 50;
Sigma = 0.20;

DividendType = {'cash'};
DividendAmounts = [0.50, 0.50, 0.50, 0.50];
ExDividendDates = {'03-Jan-2008', '01-Apr-2008', '05-July-2008', '01-Oct-2008'};

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

StockSpec =

             FinObj: 'StockSpec'
              Sigma: 0.2000
         AssetPrice: 50
       DividendType: {'cash'}
    DividendAmounts: [0.5000 0.5000 0.5000 0.5000]
```

```
    ExDividendDates: [733410 733499 733594 733682]
```

Examine the StockSpec structure:

```
datedisp(StockSpec.ExDividendDates)
03-Jan-2008   01-Apr-2008   05-Jul-2008   01-Oct-2008

StockSpec.DividendType

ans =

    'cash'
```

The StockSpec structure encapsulates the information of the stock and its four cash dividends.

Example 2. Consider two stocks that are trading at $40 and $35. The first one provides two cash dividends of $0.25 on March 1, 2008 and June 1, 2008. The second stock provides a continuous dividend yield of 3%. The stocks have a volatility of 30% per annum. Using this data, create the structure StockSpec:

```
AssetPrice = [40; 35];
Sigma = .30;

DividendType = {'cash'; 'continuous'};
DividendAmount = [0.25, 0.25 ; 0.03 NaN];

DividendDate1 = 'March-01-2008';
DividendDate2 = 'Jun-01-2008';

StockSpec = stockspec(Sigma, AssetPrice, DividendType, DividendAmount,...
{ DividendDate1, DividendDate2 ; NaN NaN})
StockSpec =

            FinObj: 'StockSpec'
             Sigma: [2x1 double]
        AssetPrice: [2x1 double]
```

```
      DividendType: {2x1 cell}
    DividendAmounts: [2x2 double]
    ExDividendDates: [2x2 double]
```

Examine the `StockSpec` structure:

```
datedisp(StockSpec.ExDividendDates)
01-Mar-2008   01-Jun-2008
   NaN            NaN

StockSpec.DividendType

ans =

    'cash'
    'continuous'
```

The `StockSpec` structure encapsulates the information of the two stocks and their dividends.

See Also

`crrprice`, `crrtree`, `intenvset`, `optstockbybjs`, `optstockbyblk`, `optstockbybls`, `optstockbyrgw`

Purpose

Calculate price of supershare digital options using Black-Scholes model

Syntax

```
Price = supersharebybls(RateSpec, StockSpec, Settle, Maturity,
OptSpec, StrikeLow, StrikeHigh)
```

Arguments

`RateSpec`	The annualized, continuously compounded rate term structure. For information on the interest rate specification, see `intenvset`.
`StockSpec`	Stock specification. See `stockspec`.
`Settle`	`NINST`-by-`1` vector of settlement or trade dates.
`Maturity`	`NINST`-by-`1` vector of maturity dates.
`StrikeLow`	`NINST`-by-`1` vector of low strike price values.
`StrikeHigh`	`NINST`-by-`1` vector of high strike price values.

Description

`Price = supersharebybls(RateSpec, StockSpec, Settle, Maturity, OptSpec, StrikeLow, StrikeHigh)` computes supershare digital option prices using the Black-Scholes model.

`Price` is a `NINST`-by-`1` vector of expected option prices.

Examples

Consider a supershare based on a portfolio of nondividend paying stocks with a lower strike of 350 and an upper strike of 450. The value of the portfolio on November 1, 2008 is 400. The risk-free rate is 4.5% and the volatility is 18%. Using this data, calculate the price of the supershare option on February 1, 2009.

Create the `RateSpec`:

```
Settle = 'Nov-1-2008';
Maturity = 'Feb-1-2009';
```

```
Rates = 0.045;
Basis = 1;
Compounding = -1;
RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle,...
'EndDates', Maturity, 'Rates', Rates, 'Compounding', Compounding, 'Basis', Basis);
```

Define the `StockSpec`:

```
AssetPrice = 400;
Sigma = .18;
StockSpec = stockspec(Sigma, AssetPrice);
```

Define the high and low strike points:

```
StrikeLow = 350;
StrikeHigh = 450;
```

Calculate the price:

```
Pssh = supersharebybls(RateSpec, StockSpec, Settle, Maturity,...
StrikeLow, StrikeHigh)

Pssh =

    0.9411
```

See Also

`assetbybls`, `cashbybls`, `gapbybls`, `supersharesensbybls`

Purpose

Calculate price and sensitivities of supershare digital options using Black-Scholes model

Syntax

```
PriceSens = supersharesensbybls(RateSpec, StockSpec, Settle,
Maturity, StrikeLow, StrikeHigh)
PriceSens = supersharesensbybls(RateSpec, StockSpec, Settle,
Maturity, StrikeLow, StrikeHigh, OutSpec)
```

Arguments

`RateSpec`	The annualized, continuously compounded rate term structure. For information on the interest rate specification, see `intenvset`.
`StockSpec`	Stock specification. See `stockspec`.
`Settle`	`NINST`-by-`1` vector of settlement or trade dates.
`Maturity`	`NINST`-by-`1` vector of maturity dates.
`StrikeLow`	`NINST`-by-`1` vector of low strike price values.
`StrikeHigh`	`NINST`-by-`1` vector of high strike price values.
`OutSpec`	(Optional) All optional inputs are specified as matching parameter name/value pairs. The parameter name is specified as a character string, followed by the corresponding parameter value. You can specify parameter name/value pairs in any order. Names are case-insensitive and partial string matches are allowed provided no ambiguities exist. Valid parameter names are: • `NOUT`-by-`1` or `1`-by-`NOUT` cell array of strings indicating the nature and order of the outputs for the function. Possible values are `'Price'`, `'Delta'`, `'Gamma'`, `'Vega'`, `'Lambda'`, `'Rho'`, `'Theta'`, or `'All'`.

For example, `OutSpec = {'Price'; 'Lamba'; 'Rho'}` specifies that the output should be `Price`, `Lambda`, and `Rho`, in that order.

To invoke from a function: `[Price, Lambda, Rho] = supersharesensbybls(..., 'OutSpec', {'Price', 'Lamba', 'Rho'})`

`OutSpec = {'All'}` specifies that the output should be `Delta`, `Gamma`, `Vega`, `Lambda`, `Rho`, `Theta`, and `Price`, in that order. This is the same as specifying `OutSpec` as `OutSpec = {'Delta', 'Gamma', 'Vega', 'Lambda', 'Rho', 'Theta', 'Price'};`.

- Default is `OutSpec = {'Price'}`.

Description

`PriceSens = supersharesensbybls(RateSpec, StockSpec, Settle, Maturity, StrikeLow, StrikeHigh)` computes supershare option prices using the Black-Scholes option pricing model.

`PriceSens = supersharesensbybls(RateSpec, StockSpec, Settle, Maturity, StrikeLow, StrikeHigh, OutSpec)` includes an `OutSpec` argument defined as parameter/value pairs, and computes supershare option prices and sensitivities using the Black-Scholes option pricing model.

`PriceSens` is a `NINST`-by-`1` vector of expected option prices and sensitivities.

Examples

Consider a supershare based on a portfolio of nondividend paying stocks with a lower strike of 350 and an upper strike of 450. The value of the portfolio on November 1, 2008 is 400. The risk-free rate is 4.5% and the volatility is 18%. Using this data, calculate the price and sensitivity of the supershare option on February 1, 2009.

Create the `RateSpec`:

```
Settle = 'Nov-1-2008';
Maturity = 'Feb-1-2009';
Rates = 0.045;
Basis = 1;
Compounding = -1;
RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle,...
'EndDates', Maturity, 'Rates', Rates, 'Compounding', Compounding, 'Basis', Basis);
```

Define the `StockSpec`:

```
AssetPrice = 400;
Sigma = .18;
StockSpec = stockspec(Sigma, AssetPrice);
```

Define the high and low strike points:

```
StrikeLow = 350;
StrikeHigh = 450;
```

Calculate the price:

```
Pssh = supersharebybls(RateSpec, StockSpec, Settle, Maturity,...
StrikeLow, StrikeHigh)

Pssh =

    0.9411
```

Compute the delta and theta of the supershare option:

```
OutSpec = { 'delta';'theta'};
[Delta, Theta]= supersharesensbybls(RateSpec, StockSpec, Settle,...
Maturity, StrikeLow, StrikeHigh, 'OutSpec', OutSpec)

Delta =

   -0.0010
```

```
Theta =

   -1.0102
```

See Also

supersharebybls

Purpose

Price swap instrument from BDT interest-rate tree

Syntax

```
[Price, PriceTree, CFTree, SwapRate] = swapbybdt(BDTTree,
LegRate, Settle, Maturity, LegReset, Basis, Principal, LegType,
Options, EndMonthRule)
```

Arguments

`BDTTree`	Interest-rate tree structure created by `bdttree`.
`LegRate`	Number of instruments (`NINST`)-by-`2` matrix, with each row defined as: `[CouponRate Spread]` or `[Spread CouponRate]` `CouponRate` is the decimal annual rate. `Spread` is the number of basis points over the reference rate. The first column represents the receiving leg, while the second column represents the paying leg.
`Settle`	Settlement date. `NINST`-by-`1` vector of serial date numbers or date strings. `Settle` must be earlier than `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.

- 0 = actual/actual (default)
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

`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`.
`EndMonthRule`	(Optional) `NINST`-by-`1` vector representing the End-of-month rule. Default = `1`.

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(BDTTree, LegRate, Settle, Maturity, LegReset, Basis, Principal, LegType, EndMonthRule)` 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 `NaN`s 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}
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

swapbybk

Purpose

Price swap instrument from Black-Karasinski interest-rate tree

Syntax

```
[Price, PriceTree, CFTree, SwapRate] = swapbybk(BKTree,
LegRate, Settle, Maturity, LegReset, Basis, Principal,
LegType, EndMonthRule)
```

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 `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.

- 0 = actual/actual (default)
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

`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`.
`EndMonthRule`	(Optional) `NINST`-by-`1` vector representing the End-of-month rule. Default = `1`.

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, EndMonthRule)` 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 `BKTree` included in the MAT-file `deriv.mat`. The `BKTree` 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(BKTree, 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(BKTree, LegRate, ...
Settle, Maturity, LegReset, Basis, Principal, LegType)

Price =
```

```
     0

PriceTree =

     FinObj: 'BKPriceTree'
       PTree: {1x5 cell}
        tObs: [0 1 2 3 4]
     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

Purpose

Price swap instrument from HJM interest-rate tree

Syntax

```
[Price, PriceTree, CFTree, SwapRate] = swapbyhjm(HJMTree,
LegRate, Settle, Maturity, LegReset, Basis, Principal,
LegType, EndMonthRule)
```

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 `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. • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ICMA) • 9 = actual/360 (ICMA) • 10 = actual/365 (ICMA) • 11 = 30/360E (ICMA) • 12 = actual/actual (ISDA) • 13 = BUS/252
`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`.
`EndMonthRule`	(Optional) `NINST`-by-`1` vector representing the End-of-month rule. Default = `1`.

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, EndMonthRule)` 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;
```

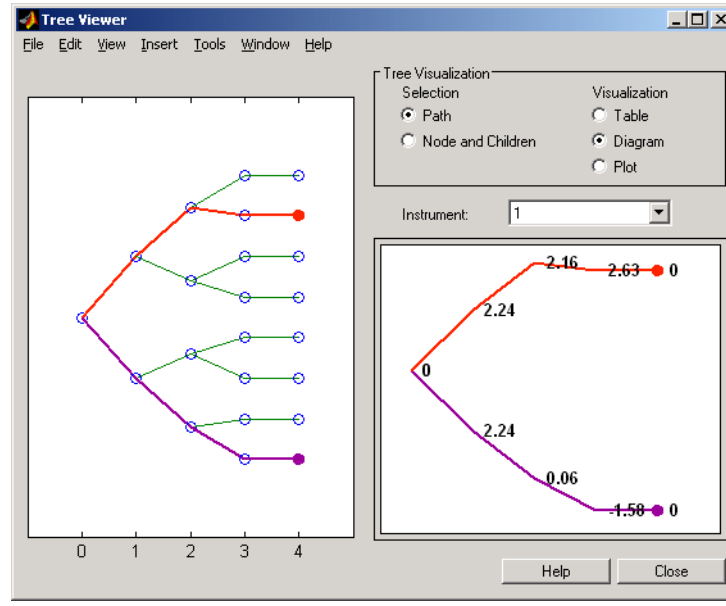
Use `swapbyhjm` to compute the price of the swap.

```
[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, EndMonthRule)
```

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 `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.

- 0 = actual/actual (default)
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

`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`.
`EndMonthRule`	(Optional) `NINST`-by-`1` vector representing the End-of-month rule. Default = `1`.

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, EndMonthRule)` 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 `HWTree` included in the MAT-file `deriv.mat`. The `HWTree` 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(HWTree, LegRate, ...
Settle, Maturity, LegReset, Basis, Principal, LegType)

Price =

    5.9109

PriceTree =

     FinObj: 'HWPriceTree'
      PTree: {1x5 cell}
       tObs: [0 1 2 3 4]
    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(HWTree, LegRate, ...
Settle, Maturity, LegReset, Basis, Principal, LegType)
Price =

   1.4211e-014

PriceTree =

FinObj: 'HWPriceTree'
       PTree: {1x5 cell}
        tObs: [0 1 2 3 4]
     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, EndMonthRule)
```

Arguments

`RateSpec`	A structure containing the properties of an interest-rate structure. See `intenvset` for information on creating `RateSpec`.
`LegRate`	Number of instruments (`NINST`)-by-`2` matrix, with each row defined as: `[CouponRate Spread]` or `[Spread CouponRate]` `CouponRate` is the decimal annual rate. `Spread` is the number of basis points over the reference rate. The first column represents the receiving leg, while the second column represents the paying leg.
`Settle`	Settlement date. `NINST`-by-`1` vector of serial date numbers or date strings representing the settlement date for each swap. `Settle` must be earlier than `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.

- 0 = actual/actual (default)
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

`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.

`EndMonthRule` (Optional) `NINST`-by-`1` vector representing the End-of-month rule. Default = `1`.

Description

`[Price, SwapRate] = swapbyzero(RateSpec, LegRate, Settle, Maturity, LegReset, Basis, Principal, LegType, EndMonthRule)` 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)

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

Price swaption from BDT interest-rate tree

Syntax

```
[Price, PriceTree] = swaptionbybdt(BDTTree, OptSpec, Strike,
ExerciseDates, Spread, Settle, Maturity,
'Name1', Value1, 'Name2', Value2)
```

Arguments

`BDTTree`	Interest-rate tree structure created by `bdttree`.
`OptSpec`	`NINST`-by-1 cell array of strings `'call'` or `'put'`. A call swaption entitles the buyer to pay the fixed rate. A put swaption entitles the buyer to receive the fixed rate.
`Strike`	`NINST`-by-1 vector for strike swap rate values.
`ExerciseDates`	For a European option: `NINST`-by-1 vector of exercise dates. Each row is the schedule for one option. For a European option, there is only one `ExerciseDate` on 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 swap `Settle` and the single listed `ExerciseDate`.
`Spread`	`NINST`-by-1 vector representing the number of basis points over the reference rate.
`Settle`	`NINST`-by-1 vector of dates representing the settle date for each swap.

`Maturity`	`NINST`-by-`1` vector of dates representing the maturity date for each swap.

Note All optional inputs that follow are specified as matching parameter name/value pairs. The parameter name is specified as a character string, followed by the corresponding parameter value. Parameter name/value pairs may be specified in any order; names are case-insensitive and partial string matches are allowed provided no ambiguities exist.

`AmericanOpt`	(Optional) `NINST`-by-`1` flags options: • 0 for European options • 1 for American options
`SwapReset`	(Optional) `NINST`-by-`1` vector representing the reset frequency per year for the underlying swap. Default is 1.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ICMA)

	• 9 = actual/360 (ICMA) • 10 = actual/365 (ICMA) • 11 = 30/360E (ICMA) • 12 = actual/actual (ISDA) • 13 = BUS/252
`Principal`	(Optional) `NINST`-by-1 vector of the notional principal amounts. Default is 100.
`Options`	(Optional) Derivatives pricing options structure created with `derivset`.

Description

`[Price, PriceTree] = swaptionbybdt(BDTTree, OptSpec, Strike,ExerciseDates, Spread, Settle, Maturity,'Name1', Value1, 'Name2', Value2)` computes the price of a swaption from a BDT interest-rate tree. The swaption may be a call swaption or a put swaption.

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

A call swaption or payer swaption allows the option buyer to enter into an interest rate swap in which the buyer of the option pays the fixed rate and receives the floating rate.

A put swaption or receiver swaption allows the option buyer to enter into an interest rate swap in which the buyer of the option receives the fixed rate and pays the floating rate.

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

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

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

Examples

Price a 5-year call swaption using a BDT interest-rate tree.

Assume that interest rate and volatility are fixed at 6% and 20% annually between the valuation date of the tree until its maturity. Build a tree with the following data:

```
Rates = 0.06 * ones (10,1);
StartDates = ['jan-1-2007';'jan-1-2008';'jan-1-2009';'jan-1-2010';'jan-1-2011';...
'jan-1-2012';'jan-1-2013';'jan-1-2014';'jan-1-2015';'jan-1-2016'];

EndDates =['jan-1-2008';'jan-1-2009';'jan-1-2010';'jan-1-2011';'jan-1-2012';...
'jan-1-2013';'jan-1-2014';'jan-1-2015';'jan-1-2016';'jan-1-2017'];
ValuationDate = 'jan-1-2007';
Compounding = 1;
```

Determine the `RateSpec`:

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

Use `VolSpec` to compute the interest rate volatility:

```
Volatility = 0.20 * ones (10,1);  VolSpec = bdtvolspec(ValuationDate,...
EndDates, Volatility);
```

Use `TimeSpec` to specify the structure of the time layout for an equal probabilities tree:

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

Build the BDT tree:

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

Use the following swaption arguments:

```
SwapSettlement = 'jan-1-2007';
SwapMaturity   = 'jan-1-2015';
Spread = 0;
SwapReset = 1;
Principal = 100;
OptSpec = 'call';
Strike=.062;
ExerciseDates = 'jan-1-2012';
Basis=1;
```

Price the swaption

```
[Price, PriceTree] = swaptionbybdt(BDTTree, OptSpec, Strike, ExerciseDates, ...
Spread, SwapSettlement, SwapMaturity, 'SwapReset', SwapReset, ...
'Basis', Basis, 'Principal', Principal)
```

to return

```
Price =        2.0592

PriceTree =
FinObj: 'BDTPriceTree'
tObs: [0 1 2 3 4 5 6 7 8 9 10]
PTree: {1x11 cell}
```

See Also

bdttree, instswaption, swapbybdt

swaptionbybk

Purpose

Price swaption from BK interest-rate tree

Syntax

```
[Price, PriceTree] = swaptionbybk(BKTTree, OptSpec, Strike,
ExerciseDates, Spread, Settle, Maturity,
'Name1', Value1, 'Name2', Value2)
```

Arguments

`BKTree`	Interest-rate tree structure created by `bktree`.
`OptSpec`	`NINST`-by-1 cell array of strings `'call'` or `'put'`. A call swaption entitles the buyer to pay the fixed rate. A put swaption entitles the buyer to receive the fixed rate.
`Strike`	`NINST`-by-1 vector for strike swap rate values.
`ExerciseDates`	For a European option: `NINST`-by-1 vector of exercise dates. Each row is the schedule for one option. For a European option, there is only one `ExerciseDate` on 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 swap `Settle` and the single listed `ExerciseDate`.
`Spread`	`NINST`-by-1 vector representing the number of basis points over the reference rate.
`Settle`	`NINST`-by-1 vector of dates representing the settle date for each swap.

`Maturity`	`NINST`-by-`1` vector of dates representing the maturity date for each swap.

Note All optional inputs that follow are specified as matching parameter name/value pairs. The parameter name is specified as a character string, followed by the corresponding parameter value. Parameter name/value pairs may be specified in any order; names are case-insensitive and partial string matches are allowed provided no ambiguities exist.

`AmericanOpt`	(Optional) `NINST`-by-`1` flags options: • 0 for European options • 1 for American options
`SwapReset`	(Optional) `NINST`-by-`1` vector representing the reset frequency per year for the underlying swap. Default is 1.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ICMA)

	• 9 = actual/360 (ICMA) • 10 = actual/365 (ICMA) • 11 = 30/360E (ICMA) • 12 = actual/actual (ISDA) • 13 = BUS/252
`Principal`	(Optional) `NINST`-by-`1` vector of the notional principal amounts. Default is 100.
`Options`	(Optional) Derivatives pricing options structure created with `derivset`.

Description

`[Price, PriceTree] = swaptionbybk(BKTTree, OptSpec, Strike,ExerciseDates, Spread, Settle, Maturity,'Name1', Value1, 'Name2', Value2)` computes the price of a swaption from a BK interest-rate tree.

Note The `Settle` date for every swaption is set to the `ValuationDate` of the BK tree. The swap argument `Settle` is ignored.

The swaption may be a call swaption or a put swaption.

A call swaption or payer swaption allows the option buyer to enter into an interest rate swap in which the buyer of the option pays the fixed rate and receives the floating rate.

A put swaption or receiver swaption allows the option buyer to enter into an interest rate swap in which the buyer of the option receives the fixed rate and pays the floating rate.

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

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

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

Examples

Price a 4-year call and put swaption using a BK interest-rate tree with the following data.

Specify the `RateSpec`, assuming the interest rate is fixed at 7% annually:

```
 Rates =0.07 * ones (10,1);
Compounding = 2;
StartDates = ['jan-1-2007';'jul-1-2007';'jan-1-2008';'jul-1-2008';'jan-1-2009'; ...
'jul-1-2009'; 'jan-1-2010'; 'jul-1-2010';'jan-1-2011';'jul-1-2011'];
EndDates =['jul-1-2007';'jan-1-2008';'jul-1-2008';'jan-1-2009';'jul-1-2009'; ...
'jan-1-2010'; 'jul-1-2010';'jan-1-2011';'jul-1-2011';'jan-1-2012'];
ValuationDate = 'jan-1-2007';
RateSpec = intenvset('Rates', Rates, 'StartDates', StartDates, 'EndDates', EndDates,...
'Compounding', Compounding);
```

Use `BKVolSpec` to compute the interest rate volatility:

```
Volatility = 0.10*ones(10,1);
AlphaCurve = 0.05*ones(10,1);
AlphaDates = EndDates;
BKVolSpec = bkvolspec(ValuationDate, EndDates, Volatility, AlphaDates, AlphaCurve);
```

Use `BKTimeSpec` to specify the structure of the time layout for the BK interest-rate tree.

```
BKTimeSpec = bktimespec(ValuationDate, EndDates, Compounding);
```

Build the BK tree:

```
BKTree = bktree(BKVolSpec, RateSpec, BKTimeSpec);
```

Use the following arguments for a 5-year swap and 4-year swaption:

```
SwapSettlement = 'jan-1-2007';
SwapMaturity   = 'jan-1-2012';
Spread = 0;
SwapReset = 2 ;
Principal = 100;
OptSpec = {'call' ;'put'};
Strike= [ 0.07 ; 0.0725];
ExerciseDates = 'jan-1-2011';
Basis=1;
```

Price the swaption

```
PriceSwaption = swaptionbybk(BKTree, OptSpec, Strike, ExerciseDates, ...
Spread, SwapSettlement, SwapMaturity, 'SwapReset', SwapReset, 'Basis', Basis, ...
'Principal', Principal)
```

to return

```
PriceSwaption =
0.3593
0.4756
```

See Also

`bktree`, `instswaption`, `swapbybk`

Purpose

Price swaption from HJM interest-rate tree

Syntax

```
[Price, PriceTree] = swaptionbyhjm(HJMTree, OptSpec, Strike,
ExerciseDates, Spread, Settle, Maturity,
'Name1', Value1, 'Name2', Value2)
```

Arguments

`HJMTree`	Interest-rate tree structure created by `hjmtree`.
`OptSpec`	`NINST`-by-1 cell array of strings `'call'` or `'put'`. A call swaption entitles the buyer to pay the fixed rate. A put swaption entitles the buyer to receive the fixed rate.
`Strike`	`NINST`-by-1 vector of strike swap rate values.
`ExerciseDates`	For a European option: `NINST`-by-1 vector of exercise dates. Each row is the schedule for one option. For a European option, there is only one `ExerciseDate` on 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 swap `Settle` and the single listed `ExerciseDate`.
`Spread`	`NINST`-by-1 vector representing the number of basis points over the reference rate.
`Settle`	`NINST`-by-1 vector of dates representing the settle date for each swap.

`Maturity`	`NINST`-by-`1` vector of dates representing the maturity date for each swap.

Note All optional inputs that follow are specified as matching parameter name/value pairs. The parameter name is specified as a character string, followed by the corresponding parameter value. Parameter name/value pairs may be specified in any order; names are case-insensitive and partial string matches are allowed provided no ambiguities exist.

`AmericanOpt`	(Optional) `NINST`-by-`1` flags options: • 0 for European options • 1 for American options
`SwapReset`	(Optional) `NINST`-by-`1` vector representing the reset frequency per year for the underlying swap. Default is 1.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ICMA)

	• 9 = actual/360 (ICMA) • 10 = actual/365 (ICMA) • 11 = 30/360E (ICMA) • 12 = actual/actual (ISDA) • 13 = BUS/252
`Principal`	(Optional) `NINST`-by-`1` vector of the notional principal amounts. Default is 100.
`Options`	(Optional) Derivatives pricing options structure created with `derivset`.

Description

`[Price, PriceTree] = swaptionbyhjm(HJMTree, OptSpec, Strike, ExerciseDates, Spread, Settle, Maturity,'Name1', Value1, 'Name2', Value2)` computes the price of a swaption from a HJM interest-rate tree.

Note The `Settle` date for every swaption is set to the `ValuationDate` of the HJM tree. The swap argument `Settle` is ignored.

The swaption may be a call swaption or a put swaption.

A call swaption or payer swaption allows the option buyer to enter into an interest rate swap in which the buyer of the option pays the fixed rate and receives the floating rate.

A put swaption or receiver swaption allows the option buyer to enter into an interest rate swap in which the buyer of the option receives the fixed rate and pays the floating rate.

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

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

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

Examples

Price a 1-year call swaption using an HJM interest-rate tree.

Assume that interest rate is fixed at 5% annually between the valuation date of the tree until its maturity. Build a tree with the following data.

Specify the `RateSpec`:

```
Rates = [ 0.05;0.05;0.05;0.05];
StartDates = 'jan-1-2007';
EndDates =['jan-1-2008';'jan-1-2009';'jan-1-2010';'jan-1-2011'];
ValuationDate = StartDates;
Compounding = 1;
RateSpec = intenvset('Rates', Rates, 'StartDates', StartDates, 'EndDates',...
EndDates, 'Compounding', Compounding);
```

Use `VolSpec` to compute the interest rate volatility:

```
VolSpec=hjmvolspec('Constant',0.01);
```

Use `TimeSpec` to specify the structure of the time layout for the HJM interest-rate tree:

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

Build the HJM tree:

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

Use the following swaption arguments:

```
SwapSettlement = 'jan-1-2007';
SwapMaturity   = 'jan-1-2010';
```

```
Spread = [0];
SwapReset = 1;
Basis  = 1;
Principal = 100;
OptSpec = 'call';
Strike=0.05;
ExerciseDates = '01-Jan-2008';
```

Price the swaption

```
[Price, PriceTree] = swaptionbyhjm(HJMTree, OptSpec, Strike, ExerciseDates, ...
Spread, SwapSettlement, SwapMaturity,'SwapReset', SwapReset, ...
'Basis', Basis, 'Principal', Principal)
```

to return

```
Price =

    0.9296

PriceTree =

    FinObj: 'HJMPriceTree'
    tObs: [5x1 double]
    PBush: {[0.9296]  [1x1x2 double]  [1x2x2 double]  [1x4x2 double]  [0 0 0 0 0 0 0 0]}
```

See Also

hjmtree, instswaption, swapbyhjm

swaptionbyhw

Purpose

Price swaption from HW interest-rate tree

Syntax

```
[Price, PriceTree] = swaptionbyhw(HWTree, OptSpec, Strike,
ExerciseDates, Spread, Settle, Maturity,
'Name1', Value1, 'Name2', Value2)
```

Arguments

`HWTree`	Interest-rate tree structure created by `hwtree`.
`OptSpec`	`NINST`-by-1 cell array of strings `'call'` or `'put'`. A call swaption entitles the buyer to pay the fixed rate. A put swaption entitles the buyer to receive the fixed rate.
`Strike`	`NINST`-by-1 vector for strike swap rate values.
`ExerciseDates`	For a European option: `NINST`-by-1 vector of exercise dates. Each row is the schedule for one option. For a European option, there is only one `ExerciseDate` on 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 swap `Settle` and the single listed `ExerciseDate`.
`Spread`	`NINST`-by-1 vector representing the number of basis points over the reference rate.
`Settle`	`NINST`-by-1 vector of dates representing the settle date for each swap.

`Maturity`	`NINST`-by-1 vector of dates representing the maturity date for each swap.

Note All optional inputs that follow are specified as matching parameter name/value pairs. The parameter name is specified as a character string, followed by the corresponding parameter value. Parameter name/value pairs may be specified in any order; names are case-insensitive and partial string matches are allowed provided no ambiguities exist.

`AmericanOpt`	(Optional) `NINST`-by-1 flags options: • 0 for European options • 1 for American options
`SwapReset`	(Optional) `NINST`-by-1 vector representing the reset frequency per year for the underlying swap. Default is 1.
`Basis`	(Optional) Day-count basis of the instrument. A vector of integers. • 0 = actual/actual (default) • 1 = 30/360 (SIA) • 2 = actual/360 • 3 = actual/365 • 4 = 30/360 (BMA) • 5 = 30/360 (ISDA) • 6 = 30/360 (European) • 7 = actual/365 (Japanese) • 8 = actual/actual (ICMA)

	• 9 = actual/360 (ICMA) • 10 = actual/365 (ICMA) • 11 = 30/360E (ICMA) • 12 = actual/actual (ISDA) • 13 = BUS/252
`Principal`	(Optional) `NINST`-by-`1` vector of the notional principal amounts. Default is 100.
`Options`	(Optional) Derivatives pricing options structure created with `derivset`.

Description

`[Price, PriceTree] = swaptionbyhw(HWTree, OptSpec, Strike,ExerciseDates, Spread, Settle, Maturity,'Name1', Value1, 'Name2', Value2)` computes the price of a swaption from a HW interest-rate tree.

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

The swaption may be a call swaption or a put swaption.

A call swaption or payer swaption allows the option buyer to enter into an interest rate swap in which the buyer of the option pays the fixed rate and receives the floating rate.

A put swaption or receiver swaption allows the option buyer to enter into an interest rate swap in which the buyer of the option receives the fixed rate and pays the floating rate.

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

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

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

Examples

Price a 3-year put swaption using an HW interest-rate tree with the following data.

Specify the `RateSpec`:

```
Rates =0.075 * ones (10,1);
Compounding = 2;
StartDates = ['jan-1-2007';'jul-1-2007';'jan-1-2008';'jul-1-2008';'jan-1-2009';...
'jul-1-2009';'jan-1-2010'; 'jul-1-2010';'jan-1-2011';'jul-1-2011'];
EndDates =['jul-1-2007';'jan-1-2008';'jul-1-2008';'jan-1-2009';'jul-1-2009';...
'jan-1-2010';'jul-1-2010';'jan-1-2011';'jul-1-2011';'jan-1-2012'];
ValuationDate = 'jan-1-2007';
RateSpec = intenvset('Rates', Rates, 'StartDates', StartDates, 'EndDates',...
EndDates, 'Compounding', Compounding);
```

Use `HWVolSpec` to compute the interest rate volatility:

```
Volatility = 0.05*ones(10,1);
AlphaCurve = 0.01*ones(10,1);
AlphaDates = EndDates;
HWVolSpec = hwvolspec(ValuationDate, EndDates, Volatility, AlphaDates, AlphaCurve);
```

Use `HWTimeSpec` to specify the structure of the time layout for an HW interest-rate tree:

```
HWTimeSpec = hwtimespec(ValuationDate, EndDates, Compounding);
```

Build the HW tree:

```
HWTree = hwtree(HWVolSpec, RateSpec, HWTimeSpec);
```

Use the following arguments for a 5-year swap and 3-year swaption:

```
SwapSettlement = 'jan-1-2007';
SwapMaturity   = 'jan-1-2012';
Spread = 0;
SwapReset = 2 ;
Principal = 100;
OptSpec = 'put';
Strike= 0.04;
ExerciseDates = 'jan-1-2010';
Basis=1;
```

Price the swaption

```
PriceSwaption = swaptionbyhw(HWTree, OptSpec, Strike, ExerciseDates, ...
Spread, SwapSettlement, SwapMaturity,'SwapReset', SwapReset, ...
'Basis', Basis,'Principal', Principal)
```

to return

```
PriceSwaption =

    2.9081
```

See Also

hwtree, instswaption, swapbyhw

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`	(Optional) Scalar value representing the rate at which the input zero rates were compounded when annualized. Default = `2`. This argument determines the formula for the discount factors:
	`Compounding` = `1`, `2`, `3`, `4`, `6`, `12`
	`Disc = (1 + Z/F)^(-T)`, where `F` is the compounding frequency, `Z` is the zero rate, and `T` is the time in periodic units; for example, `T = F` is 1 year.
	`Compounding` = `365`
	`Disc = (1 + Z/F)^(-T)`, where `F` is the number of days in the basis year and `T` is a number of days elapsed computed by basis.
	`Compounding` = `-1`
	`Disc = exp(-T*Z)`, where `T` is time in years.

`Basis` (Optional) Day-count basis of the instrument. A vector of integers.

- 0 = actual/actual (default)
- 1 = 30/360 (SIA)
- 2 = actual/360
- 3 = actual/365
- 4 = 30/360 (BMA)
- 5 = 30/360 (ISDA)
- 6 = 30/360 (European)
- 7 = actual/365 (Japanese)
- 8 = actual/actual (ICMA)
- 9 = actual/360 (ICMA)
- 10 = actual/365 (ICMA)
- 11 = 30/360E (ICMA)
- 12 = actual/actual (ISDA)
- 13 = BUS/252

`EndMonthRule` (Optional) End-of-month rule. A vector. This rule applies only when `Maturity` is an end-of-month date for a month having 30 or fewer days. `0` = ignore rule, meaning that a bond's coupon payment date is always the same numerical day of the month. `1` = set rule on (default), meaning that a bond's coupon payment date is always the last actual day of the month.

Description

`Dates = time2date(Settle, 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
      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(BDTTree.FwdTree, [1 2 1])
```

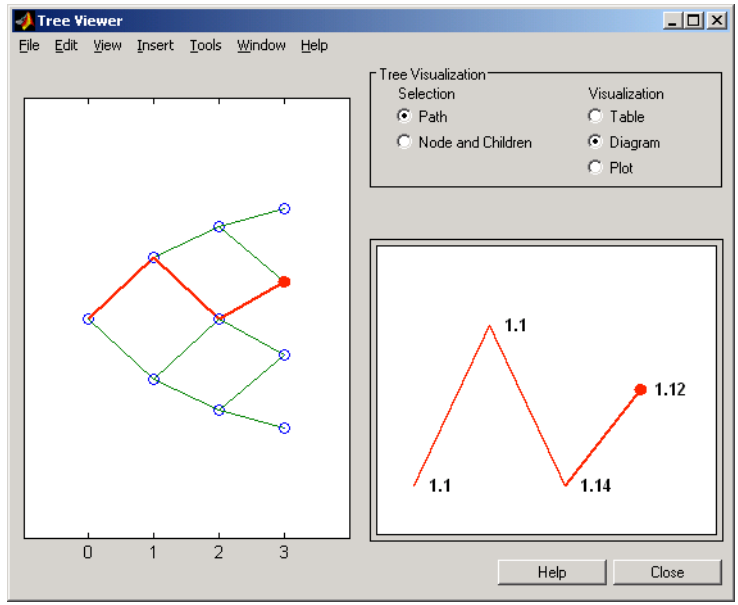
returns the rates at the tree nodes located by taking the up branch, then the down branch, and finally the up branch again.

```
FwdRates =

    1.1000
    1.0979
    1.1377
```

```
    1.1183
```

You can visualize this with the `treeviewer` function.

```
treeviewer(BDTTree)
```



See Also

`mktree`, `treeshape`

Purpose

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.

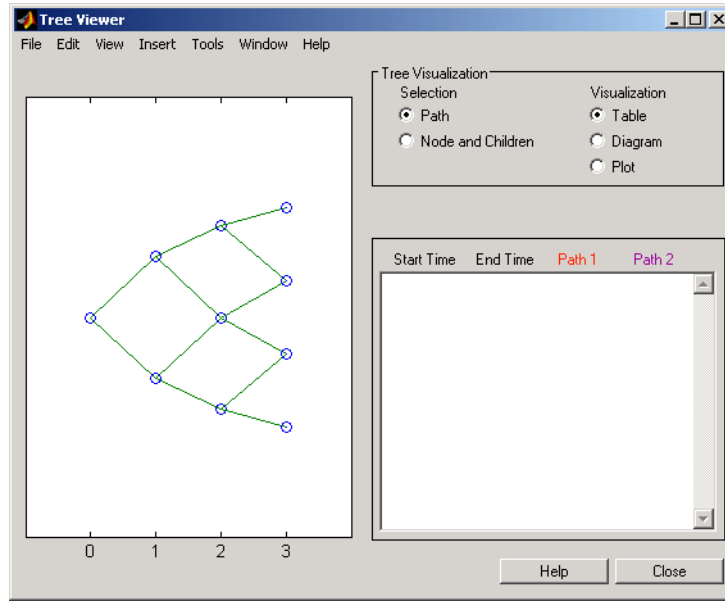
Examples

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(BDTTree)
```



With this tree

```
[NumLevels, NumPos, IsPriceTree] = treeshape(BDTTree.FwdTree)
```

returns

```
NumLevels  =
     4

NumPos     =
     1     1     1     1

IsPriceTree =
     0
```

See Also

mktree, treepath

Purpose

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 (`ITTTree`)
- 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 `ITTTree`.

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 `NaN`s 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 (`eqpprice`), or Implied Trinomial tree (`ittprice`) tree of instrument prices.
`CFTree`	`CFTree` 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 `NaN`s.)
`InstSet`	(Optional) Variable containing a collection of instruments whose prices or cash flows are contained in a tree. The collection can be created with the function `instadd` or as a cell array containing the names of the instruments. To display the names of the instruments, the field `Name` should exist in `InstSet`. If `InstSet` is not passed, `treeviewer` uses default instruments names (numbers) when displaying prices or cash flows.

Description

`treeviewer(Tree)` displays an interest rate, 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 endpoints 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

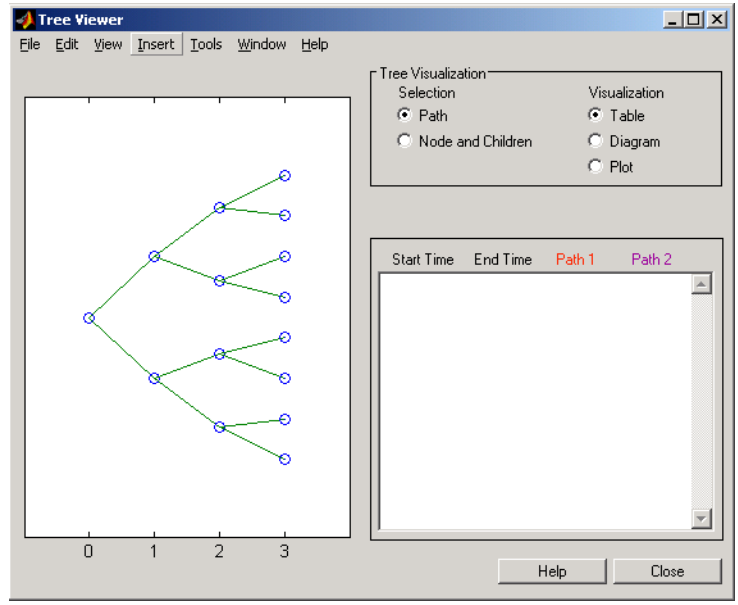
```
Parent of selected node must be selected.
```

Examples

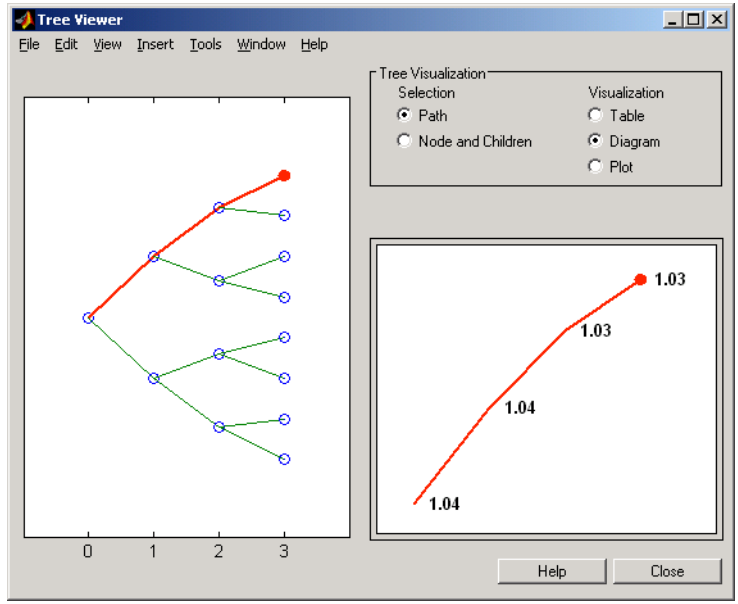
Example 1. Display an HJM Interest-Rate Tree.

```
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.

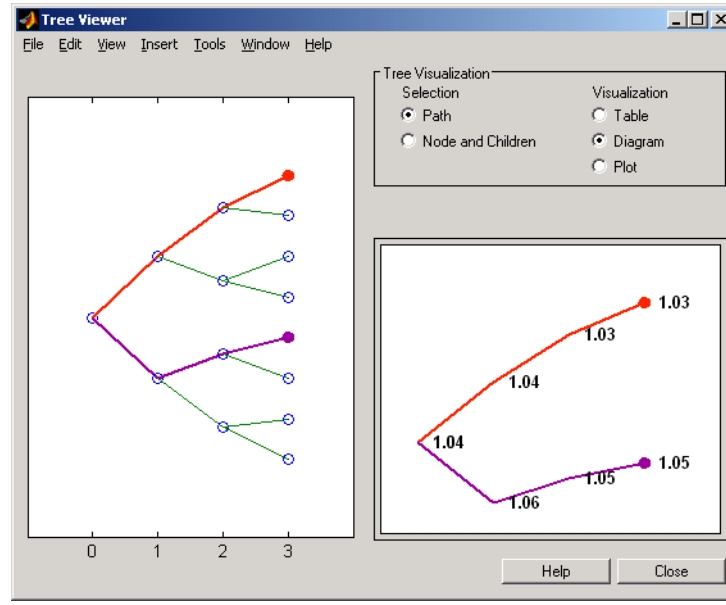


To visualize the actual interest-rate tree, go to the **Tree Visualization** pane and click on **Path** (the default) and **Diagram.** Now, select the first path by clicking on the last node (`t = 3`) of the upper branch.



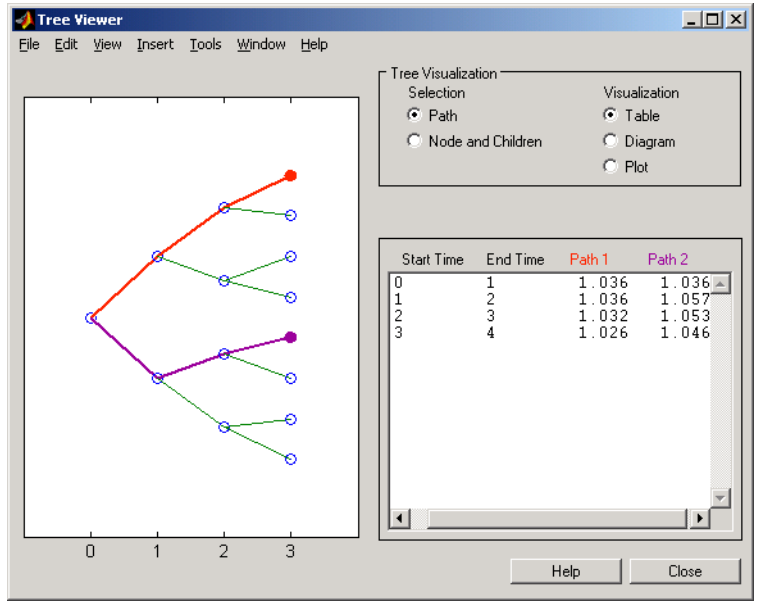
Note that the entire upper path is highlighted in red.

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.

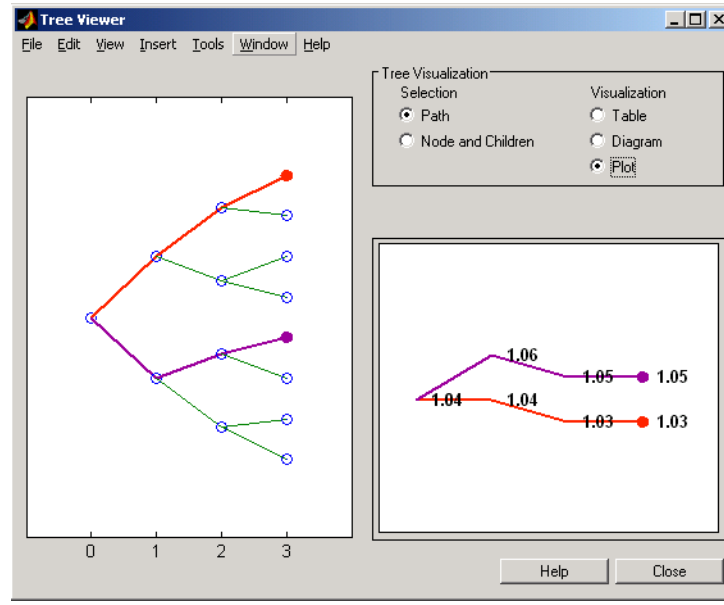


Alternative Forms of Display

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.

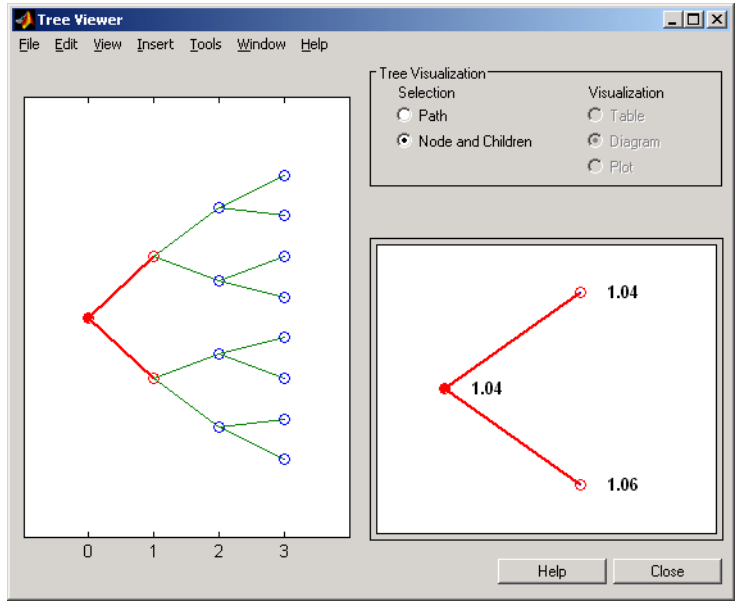


To see a plot of interest rates along the chosen branches, click **Path** and **Plot** in the **Tree Visualization** pane.



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.

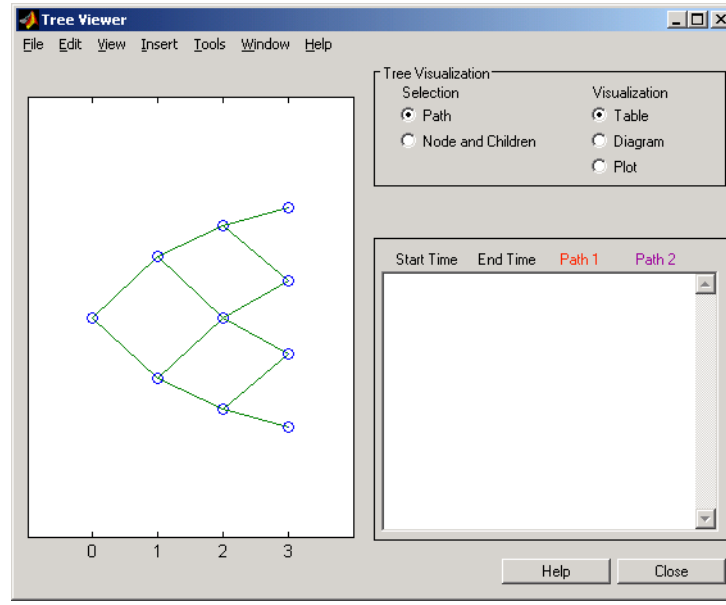


With **Node and Children** selected, the choices under **Visualization** are unavailable.

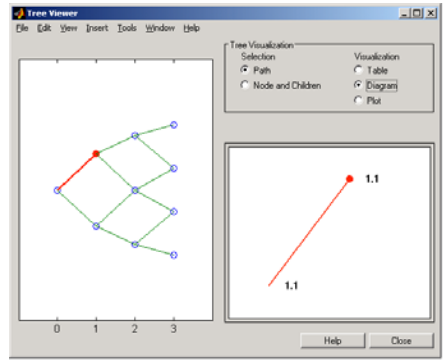
Example 2. Display a BDT Interest-Rate Tree.

```
load deriv.mat
treeviewer(BDTTree)
```

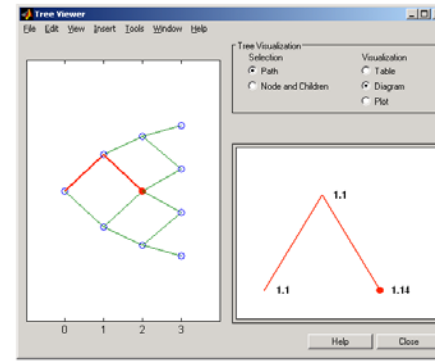
The `treeviewer` function displays the structure of a BDT tree in the left pane. The tree visualization in the right pane is blank.



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.



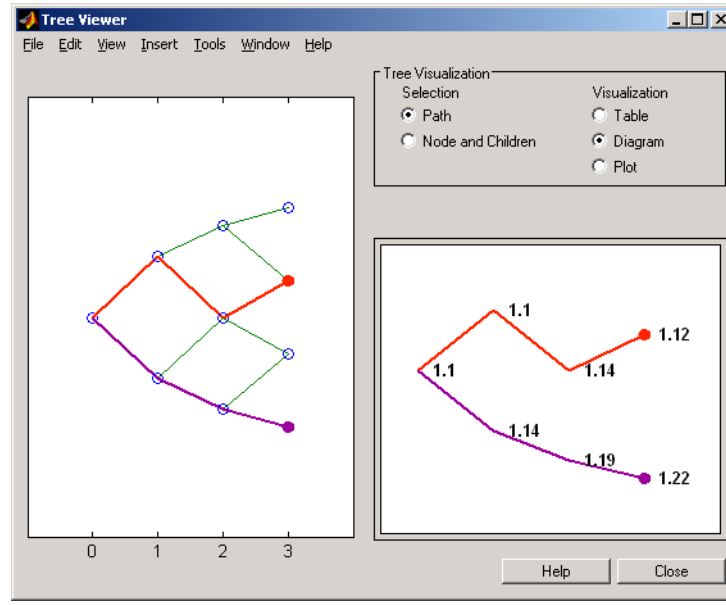
t = 1



t = 2

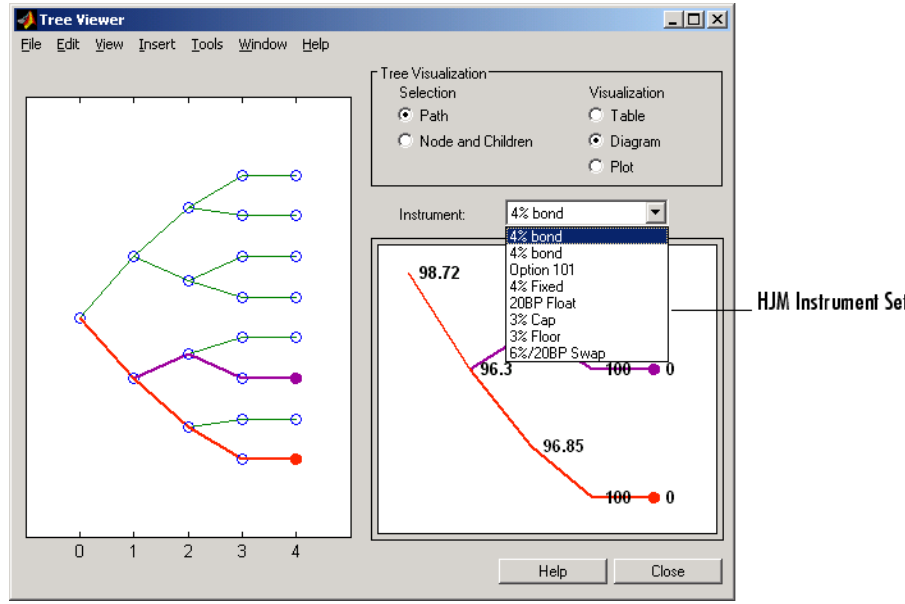
Continue clicking all nodes in succession until you reach the end of the branch. Note that the entire path you have selected is highlighted in red.

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.



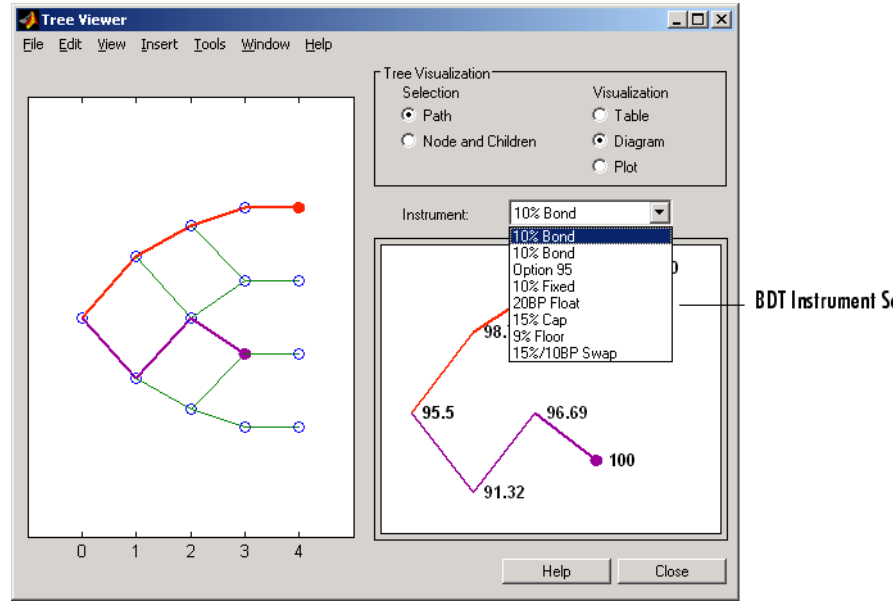
Example 3. Display an HJM Price Tree for Named Instruments.

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



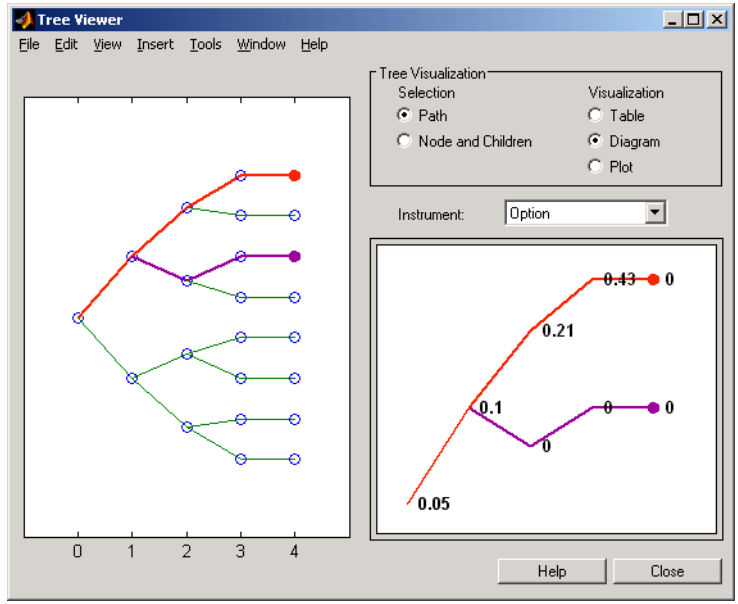
Example 4. Display a BDT Price Tree for Named Instruments.

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



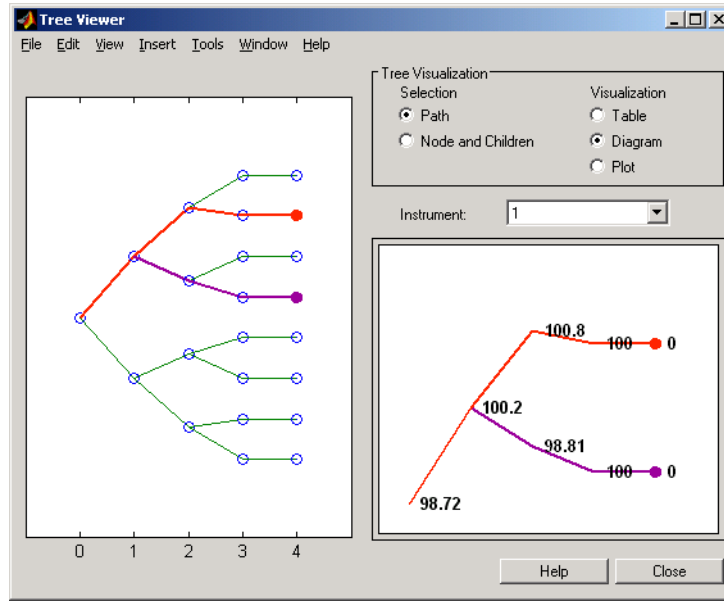
Example 5. Display an HJM Price Tree with Renamed Instruments.

```
load deriv.mat
[Price, PriceTree] = hjmprice(HJMTree, HJMInstSet);
Names = {'Bond1', 'Bond2', 'Option', 'Fixed','Float', 'Cap',...
'Floor', 'Swap'};
treeviewer(PriceTree, Names)
```



Example 6. Display an HJM Price Tree Using Default Instrument Names (Numbers).

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



See Also

`bdttree`, `bktree`, `crrtree`, `eqptreehjmtree`, `hwtree`, `instadd`, `itttreemmktbybdt`, `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])
```

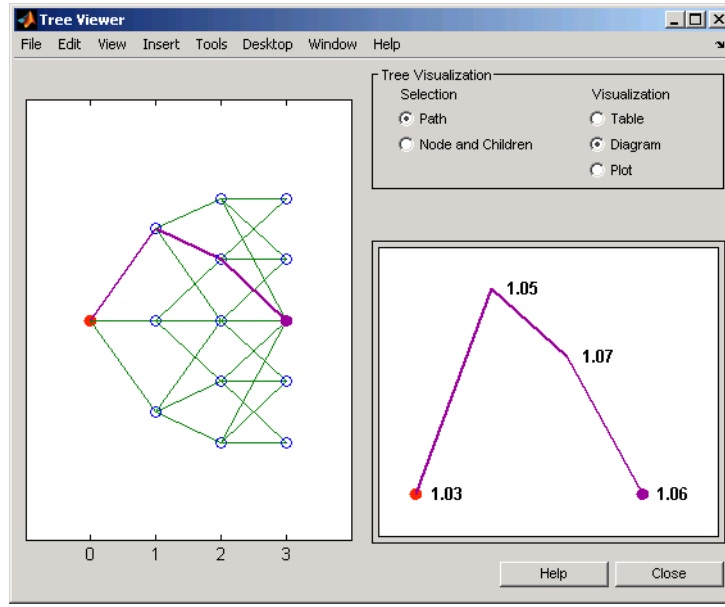
returns the rates at the tree nodes located by starting at 0, taking the up branch at the first node, the middle branch at the second node, and finally the bottom branch at the third node.

```
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.

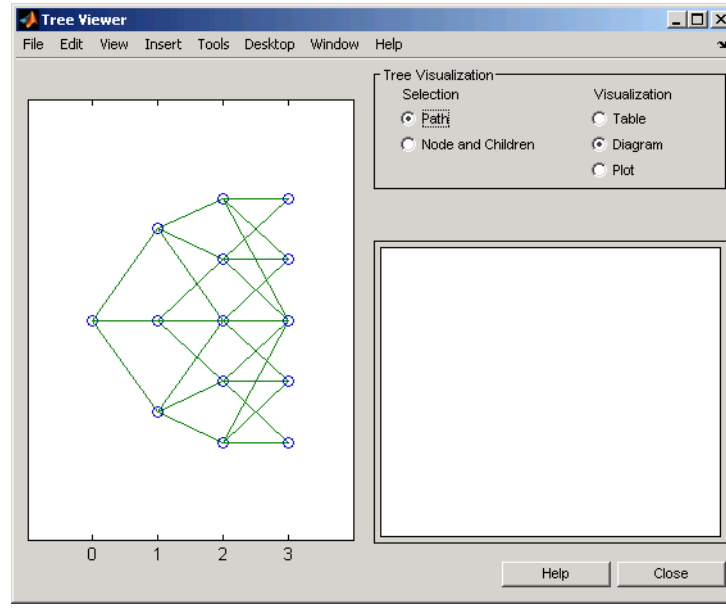
Examples

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(HWTree)
```



With this tree

```
[NumLevels, NumPos, NumStates] = trintreeshape(HWTree)
```

returns

```
NumLevels  =
     4

NumPos     =
     1     1     1     1

NumStates =
     1     3     5     5
```

See Also

`mktrintree`, `trintreepath`

A

Derivatives Pricing Options

Pricing Options Structure

In this section...

Introduction

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]
```

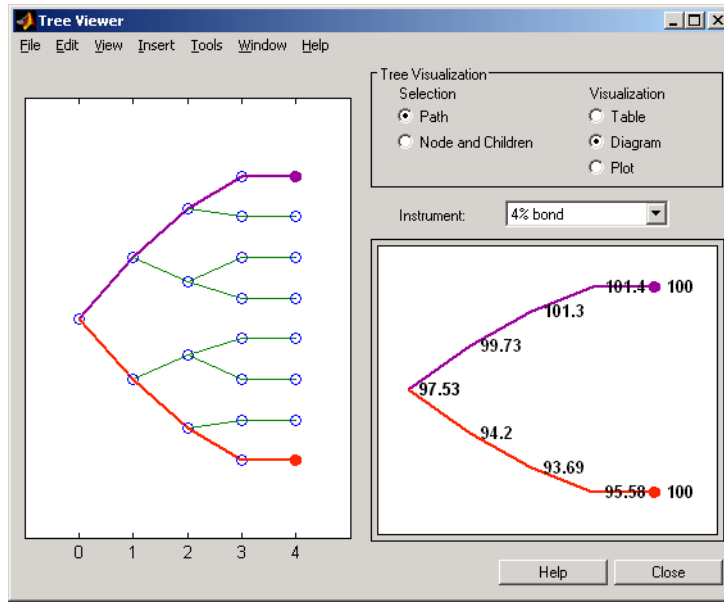
As indicated in the last warning, because the cash flows of the bond did not align with the tree dates, a new tree was generated for pricing the bond. This pricing method returns more accurate results since it guarantees that the process is arbitrage-free. It also takes longer to calculate and requires 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})
```

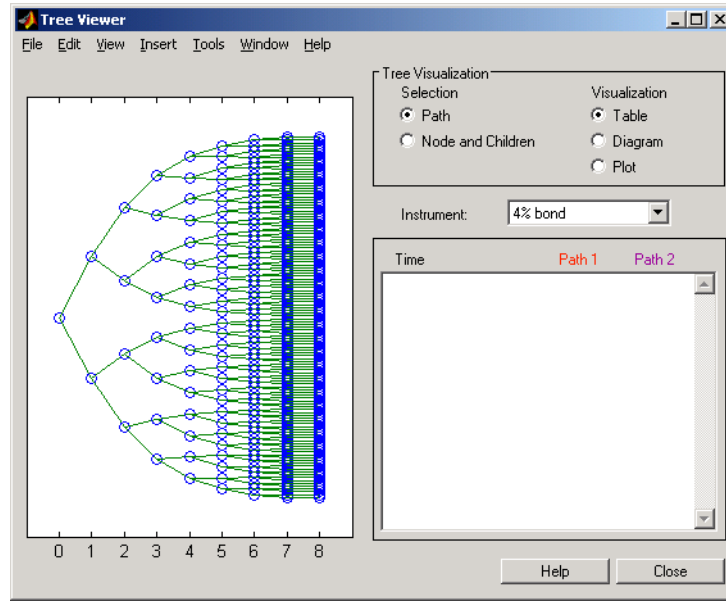
```
ans =

     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
```

B

Bibliography

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 software, 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.

Closed-Form Solutions Modeling

To learn about the Bjerksund-Stensland 2002 model, see:

Bjerksund, P. and G. Stensland, *Closed-Form Approximation of American Options*, Scandinavian Journal of Management, 1993, Vol. 9, Suppl., pp. S88-S99.

Bjerksund, P. and G. Stensland, *Closed Form Valuation of American Options*, Discussion paper 2002 (http://bora.nhh.no/bitstream/2330/711/1/bjerksund%20petter%200902.pdf).

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 software 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.

C

Examples

Use this list to find examples in the documentation.

Instrument Portfolio Examples

Interest Rate Environment Examples

HJM Examples

Volatility Modeling

BDT Examples

Rate Specification Creation

Time Specification

Sensitivity

Treeviewer Examples

Creating Equity Derivatives

Pricing Equity Derivatives

Closed-Form Solution Examples

Hedging Examples

Hedging with Constrained Portfolios

Glossary

American option
An option that can be exercised any time until its expiration date. Contrast with **European option** on page Glossary-4.

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.

asset-or-nothing option
A digital option that pays the value of the underlying security if the option expires in the money.

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-6 and **knock-out** on page Glossary-6. If the option fails to execute, the seller may pay to the purchaser a predetermined **rebate** on page Glossary-8.

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-6 and **knock-out** on page Glossary-6. If the option fails to execute, the seller may pay to the purchaser a predetermined **rebate** on page Glossary-8.

basket option
An option that provides a payoff dependent on the value of a portfolio of assets.

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-10.

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-8.

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-7.

2. A demand to submit bonds to the issuer for redemption before the maturity date.

call swaption

Allows the option buyer to enter into an interest rate swap in which the buyer of the option pays the fixed rate and receives the floating rate.

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-8.

cap

Interest-rate option that guarantees that the rate on a floating-rate loan will not exceed a certain level.

caplet

An interim cap component in a multiperiod interest-rate cap agreement.

cash-or-nothing option

A digital option that pays some fixed amount of cash if the option expires in the money.

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.

digital option

An option whose payout is fixed after the underlying stock exceeds the predetermined threshold or strike price.

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-6.

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-6.

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-10.

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.

floorlet

One of the interim period floors in a multiple period floor agreement.

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.

gap option

A digital option in which one strike decides if the option is in or out of money and another strike decides the size the size of the payoff.

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.

irregular coupon
A bond interest payment for more or less than six-months' interest. The first coupon on many bonds is irregular because payment is other than six months from the dated date.

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-10 and **down-and-in** on page Glossary-4.

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-10and **down-and-out** on page Glossary-4.

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-5.

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-8 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.

put swaption
Allows the option buyer to enter into an interest rate swap in which the buyer of the option receives the fixed rate and pays the floating rate.

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-3.

rainbow option
A single option linked to two or more underlying assets. In order for the option to pay off, all the underlying assets must move in the intended direction.

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-4.

short rate
The annualized one-period interest rate.

spot curve, spot yield curve
See **zero curve, zero-coupon yield curve** on page Glossary-11.

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.

supershare option
A digital option that pays out a proportion of the assets underlying a portfolio if the asset lies between a lower and an upper bound at the expiry of the option.

swap
A contract between two parties to exchange cash flows in the future according to some formula.

swaption
An option on an interest rate swap. It grants the option buyer the right to enter into an interest rate swap at a future date.

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-2.

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-6.

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-6.

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-9 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.

yields

The zero coupon rate.

yield curve
The zero curve.

yield volatility
The zero coupon volatilities.

zero curve, zero-coupon yield curve
A yield curve for zero-coupon bonds; zero rates versus maturity dates. Since the maturity and duration (Macaulay duration) are identical for zeros, the zero curve is a pure depiction of supply/demand conditions for loanable funds across a continuum of durations and maturities. Also known as spot curve or spot yield curve.

zero-coupon bond, or zero
A bond that, instead of carrying a coupon, is sold at a discount from its face value, pays no interest during its life, and pays the principal only at maturity.

Index

A

B

C

D

E

F

G

H

I

L

M

O

P

R

S

T

U

V